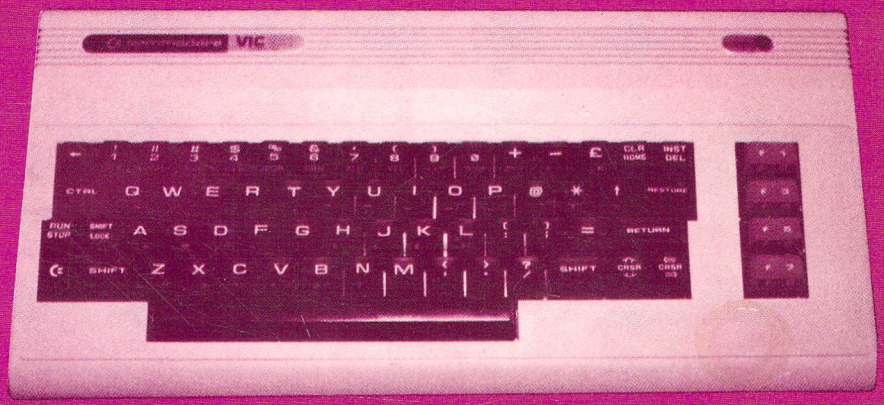
UNDERSTANDING YOUR VIC

VOLUME 1
BASIC PROGRAMMING

BREAK IN 48
READY.



by David E. Schultz

VIC is a trademark of Commodore International, Inc.

TIS

UNDERSTANDING YOUR VIC

VOLUME 1:

BASIC PROGRAMMING

David Schultz

TOTAL INFORMATION SERVICES, INC.
P.O. BOX 921
Los Alamos, NM 87544

NOTICE TO READERS

We have done our best to insure that the material in this book is accurate and reliable. However, *Total Information Services* shall have no liability or responsibility to you or any other person for inaccurancies or for the success or failure of any application to which this book might be applied.

It is good data processing procedure for the *user* to carefully test a program, to run and cross check sample sets of data, and to run the application in parallel with the previous system for a long enough period of time to insure that the results are satisfactory.

Printed in the United States of America

Copyright © 1981
Total Information Services, Inc.
P.O. Box 921
Los Alamos, New Mexico 87544

All rights reserved. Reproduction or use, without express permission, of editorial, program, or pictorial content, in any manner, is prohibited.

PREFACE

This book is for owners, prospective owners, and users of the Commodore VIC computer. It is intended to supplement the reference manuals provided by Commodore and to show how the VIC performs under various situations. It is not a BASIC programming guide, but rather a tutorial presentation on how BASIC works on the VIC.

The step-by-step exercises in this book will quickly show you how to get your VIC working for you.

The book is practical. There are many exercises already worked out. The VIC acts in certain ways which can be demonstrated; the book is written from this viewpoint rather than from explaining how someone thinks it works or should work. The book is setup for self-instruction since each topic is handled in a learn-by-doing manner.

Even though the graphics capability of the VIC is an important feature, a detailed description of graphics was not included in this book. There are two reasons for this. First, we wanted to keep the overall book at a reasonable length. Second, the graphics capability is such a large, complicated, and important topic, we felt that it deserves a book of its own.

Traditionally, computers have been considered tools for solution of mathematical problems. Computers really are information processors. In this book, we have chosen examples that minimize the use of mathematics. We did this to simplify the exercises and to demonstrate the non-mathematical applications suitable for computer solution. If you can count, you have enough mathematical background for this book.

UNDERSTANDING YOUR VIC

CONTENTS

I.	INTRODUCTION	1
	A. Assumptions Made About the User	1
	B. Exercises	1
	C. Programming	2
	D. VIC Keyboard and TV Display	4
	E. Notation	5
	F. BASIC Overview	6
II.	CBM BASIC CALCULATOR MODE	9
	A. Using Strings in CBM BASIC	10
	B. Numeric and Fractional Values	10
	C. Conversion of Data	11
	D. Balance Your Checkbook in Calculator Mode	12
	E. Reserved Words	12
	F. Modes of Variables and Constants	13
III.	INPUTTING A PROGRAM	14
	A. Blanks	14
	B. Multiple Program Statements	15
	C. Typing Mistakes	16
	D. CBM BASIC Commands	16
	E. 'STOP' and CONT	20
IV.	GETTING INFORMATION OUT OF YOUR PROGRAM	22
	A. Output Formats - Numeric Data	22
	B. Output Formats - Character Strings	24
	C. Spacing	25
V.	GETTING INFORMATION INTO YOUR PROGRAM	27
	A. Design Goals	27
	B. INPUT	27
	C. READ, DATA, and RESTORE	36
VI.	DATA REPRESENTATION	38
	A. Largest Numeric Value	38
	B. Smallest Numeric Value	39
	C. Integer Range	40
	D. Memory Space Used	41
	E. Number of Significant Digits	42
	F. Rounding	43
VII.	USING THE CASSETTE FOR PROGRAM STORAGE	44
	A. SAVE a Program	44
	B. VERIFY a Program	45
	C. LOAD a Program	47

VIII. BRANCHING 49
A. GO TO 49
B. ON X GO TO 50

IX. THE IF STATEMENT 54
A. THEN Form 54
B. GO TO Form 54
C. Multiple Statement 55

X. SUBROUTINES 58
A. GO SUB 59
B. RETURN 59
C. ON X GOSUB 60

XI. STRINGS 61
A. Legal String Names 61
B. Subscripted String Variables 62
C. Maximum String Length 63

XII. OPERATIONS ON STRINGS 64
A. Statements 64
B. Functions 65
1. ASC 65
2. CHR$ 65
3. LEFT$ 66
4. RIGHT$ 67
5. MID$ 67
6. LEN 68
7. STR$ 68
8. VAL 69

XIII. DATA REPRESENTATION AND PROCESSING PROGRAMS 70
A. Number System Conversion Programs 70
B. Logical Operations Program 72

XIV. SUBSCRIPTED VARIABLES 74
A. Legal Subscripts 75
B. Dimensions 75

XV. PROGRAM DESIGN 78
A. Psuedo Code 78
1. General Form. 79
2. Sequence. 79
3. Conditional. 79
4. Repetition. 80
5. Indentation. 81
B. Data Description 82
1. Sequence. 82
2. Selection. 82
3. Repetition. 83
C. Typical BASIC Implementation 83
1. Sequence. 84
2. Selection. 84
3. Repetition. 84

XVI. COLOR 86
A. Statement of the Problem 86
B. Refinement of the Solution 87
C. Psuedo Code of Get Choice 90
D. Psuedo Code of Set VIC 91
E. Miscellaneous Psuedo Code 91
1. Initialize. 91
2. Display Error. 92
3. Display settings. 92
F. Data Description 92
G. Writing the Program 95

XVII. SOUND 100
A. The Problem 100
B. Restrictions 100
C. Sound Program Design 101
D. Refinement of the Solution 101
E. Data Definition 104
F. Writing the Program 105

XVIII. APPENDICES

APPENDIX A: PROGRAM LISTINGS 112

A. Mailing List Program Listing 112
B. Decimal to Binary Conversion Program Listing 112
C. Binary to Decimal Conversion Program Listing 113
D. Logical Function Program Listing 114
E. Color Demonstration Program Listing 115
F. Sound Demonstration Program Listing 116

APPENDIX B: DATA PROCESSING BACKGROUND 119

A. Information 119
1. Units 119
2. Storage and Retrieval 121
B. Symbols 123
1. Arithmetic Symbols 124
2. Relational Symbols 125
C. Number Systems 126
1. Decimal-Binary-Octal-Hexadecimal 127
2. Conversions 131
D. Character Codes 134
E. Logical Operations 134
1. AND Function 135
2. Inclusive OR Function 136
3. NOT Function (Inverter) 138
F. Relational Operations 139

I. INTRODUCTION

This workbook gives a series of exercises that show the new user how to use the Commodore VIC. The most effective way to use the workbook is to sit down with a VIC and do the exercises as they are presented. Enough space has been provided in the workbook for you to add your own examples as you develop them. Later, when you need to refresh your memory on a particular topic, these examples will supply pertinent, meaningful information.

A. Assumptions Made About the User

Some VIC users are comfortable with mathematics. However, this workbook assumes that the majority are not. For that reason, most exercises will use nothing more than high school arithmetic.

This workbook also assumes you know something about the syntax and semantics of the BASIC programming language. If you do not, you should obtain one of the following books:

BASIC Programming , J. Kemeny and T. Kurtz.
BASIC for Home Computers , Albrecht, Finkel, and Brown.
What Do You Do After You Hit Return? , People's Computer Company.
Basic BASIC , James Coan.
Advanced BASIC , James Coan.
BASIC and the Personal Computer , Dwyer and Critchfield.

Read about the BASIC syntax; then alternate between this workbook and your text on BASIC. You will learn both BASIC and how to use the VIC.

B. Exercises

Some of the exercises are designed to be done by rote; that is, if you type a certain thing, then you should get a certain response. As you learn more about a feature, the exercises take the form, "What would you get if you typed this?" The next level suggests that you try various type-ins.

If you understand the topic, you should be able to anticipate most of the responses. In cases where there are idiosyncrasies, exceptions, and special problems, they are shown as "try-these" exercises.

The last level of exercises is "do-it-yourself," in

which you are given guidelines to create a personal example. If you find you cannot construct a personal example, review the previous material and try again.

You will find that you will learn more if you take the time to create or try new things. Be sure to enter the results of both successful and unsuccessful attempts in your workbook so you can have a convenient reference about what you have done. Do all the exercises in the workbook. DON'T skip any of them.

C. Programming

Contrary to what some people say, the best way to learn how to program <u>is</u> to program. You can pick up the mechanics of programming by reading books on programming, but that is like reading about moves in chess. Knowing the mechanics of chess does not make you a good chess player. Knowing the syntax of BASIC does not make you a good BASIC programmer.

Just as you can learn chess strategy by reviewing moves made by good chess players, you can learn how to program by reading other people's programs. <u>But</u> , be careful; reading bad code teaches bad habits, just as reading good code teaches good habits.

There are several useful tips all programmers should keep in mind. First, if you wish to succeed at programming, you need to develop a consistent style. To succeed at programming means to be able to write reasonably-sized codes quickly and accurately, that are easy to debug, easy to modify, and easy to use. Why is consistent style important? If you use the same approach to different problems, you can:

1. Use pieces of programs elsewhere. This saves time since you don't have to start again from scratch.

2. Understand what you have written some time ago. Consistent style minimizes questions like "What was I thinking about when I tackled this problem?"

3. Reuse documentation from previous programs.

4. Be consistent in error or warning messages.

Second, plan before you code. This is really part of your style or method of solving a problem. One temptation that many programmers succumb to is to "wing

it." That means to just sit down at the keyboard and start typing in a program without forethought, planning, or design. DON'T DO IT--not even when you know the problem is trivial. The danger in winging it on a trivial problem is that the program may work and it may work right. But, at some point you will tackle a problem thought to be trivial when it isn't. All successes at winging it will make it that much harder to admit that a particular problem is not trivial. In addition, all the winging-it successes will develop no skills at systematically designing a program and will also tempt you to slip back into that mode whenever you think you have another "trivial" problem to solve. Fig. 1 shows a good approach to writing programs.

What is a good way to improve a program? When you have finished a program (that is, you have it working as desired), discard it and start over with the design phase. Why start there? Because now that you have the program working, you know more about the problem, and you can now do a better, more concise, and more effective design. Trying to improve a program by patching up a faulty design might gain you a little, but it may introduce more problems than it removes. Remember, it is never too late to discard a program and start over. Look at your program more like a rough draft--something to learn from. Even though it has a short life, it has served its purpose if you have learned how to do a better job from it.

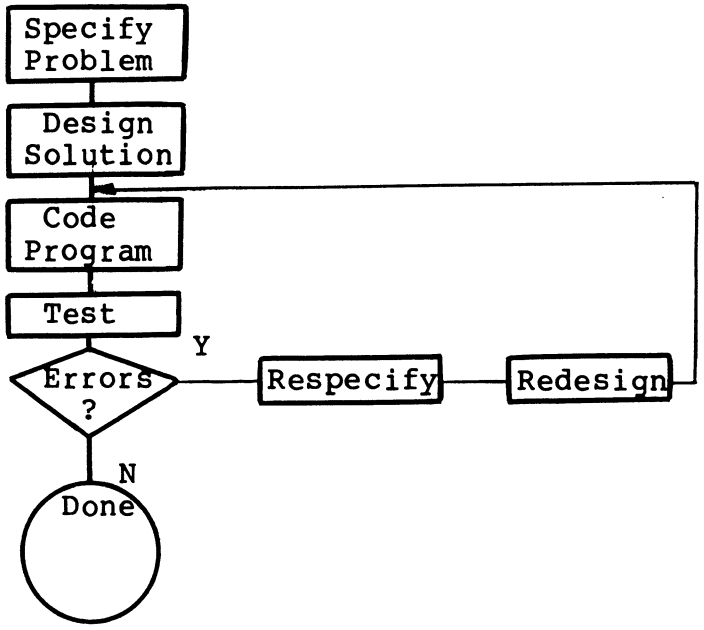


Fig. 1 Program Development Flowchart.

Many publications have stressed the idea that programming is easy and that anyone can learn to program. This is somewhat misleading. During the last 15 years, we have taught programming in different languages on different machines to people with a variety of backgrounds. Some became programmers; many did not. Our experience is not unique. Just as some people can become good bridge players, piano players, or tennis players, others do not or cannot.

However, you do not have to become an accomplished programmer to enjoy your VIC. Preprogrammed cassettes are available. If you understand some of the BASIC language, you can tailor these programs and make them more personal for your application.

D. VIC Keyboard and TV Display

The VIC keyboard (Fig. 2) and display are the primary input and output devices. To successfully communicate with your VIC, you must understand the features of the keyboard and how to use the video display effectively. You can learn this by doing exercises in the manual that CBM supplies.

The special keys on the VIC keyboard can cause some confusion. This workbook identifies the special keys with the notation 'KEYNAME'. 'KEYNAME' means press the named key.

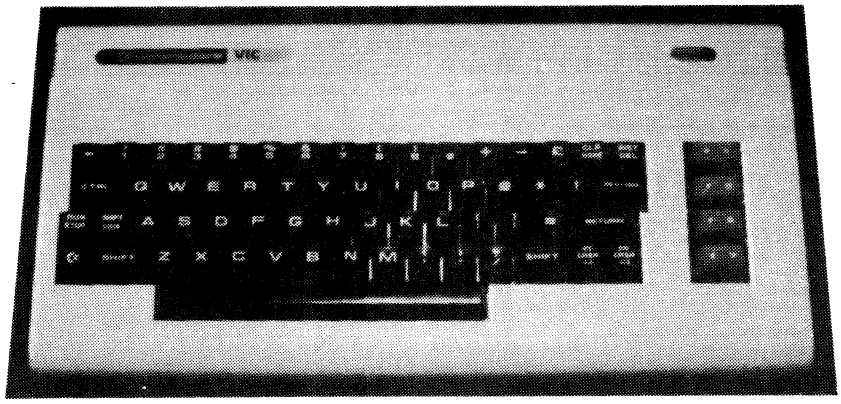


Fig. 2 The VIC Keyboard.

E. Notation

We use a consistent notation in this workbook to indicate what is to be typed on the keyboard (T:), what appears on the TV display (R:), and what indicates blanks are to be typed (b). For example:

```
T:  info
```

means to type the characters contained on the line after the colon (:) followed by a 'RETURN'.

```
R:  response
```

means that the system response to the previous T: line should be a line on the TV with the information on the R: line.

When blanks are important, they are specified by b. For example:

```
T: ?"ABbC"  ('RETURN' Key)
```

means to type ?, then ", then the letter A, then the letter B, then a space, then the letter C, then " followed by a 'RETURN'.

Turn on your VIC. You should see

```
R:  **** CBM BASIC V2 ****
R:  3583 BYTES FREE
R:  READY.
```

The number listed for free space is for a 5K RAM system.

Now let's run the example all together:

```
T: ?"ABbC"  ('RETURN' Key)

R: AB C
```

(In showing the TV display, we will not consider leading blanks. The AB C shown above may or may not be preceded by blanks on the screen.) The 'RETURN' key must be pressed at the end of each line. We will assume you know that and will not use ('RETURN' Key) in any more examples.

The special keys on the VIC keyboard can cause some confusion. This workbook identifies the special keys with the notation 'KEYNAME'. 'KEYNAME' means press the named key. The unquoted sequence of characters OTHER means press the five keys O T H E R in succession.

Example: Special key notation.

```
T:  'HOME'

R:  (Cursor should go to upper left corner).
```

means press the key labeled HOME.

```
T:  'CLR'
T:  HOME
R:  ?SYNTAX
R:   ERROR
R:  READY.
```

means press the four keys H O M E in succession.

F. BASIC Overview

BASIC has three classes of capabilities: commands, statements, and functions.

Commands are facilities that might be considered as part of the operating system or environment. Commands handle or manipulate global items, such as programs.

Statements are elements of the program itself. Statements are made up of keywords, variables, constants, operators, and functions.

A user-defined function is a BASIC statement that returns a single value each time it is referenced. Another type of function is the library function. The VIC has several library (built-in) functions, such as SIN, RND, LOG, INT, etc. These functions are referenced by their names.

1. Constants . BASIC programs are made up of statements that contain keywords, variables, operators, and constants. The VIC can have two kinds of constants. Numeric constants are numbers with or without a decimal point. If a number has a decimal point it is sometimes called a floating point number. A number without a decimal point is often called an integer. Floating point is used to represent a wider range of numbers than integer (see Section VI on data representation for more details). Even though there are two types of numeric constants, the VIC represents them the same way, i.e., as floating point. An integer constant, like 5, is represented as 5.0.

The other kind of constant that you can have in the VIC is a character string. A character string constant is signalled by quote (") marks. The symbols "AB" mean the string of two characters, A followed by B.

2. Variables . Variables are "names" that may take on different values during a problem. In BASIC the first character in a variable must be alphabetic and not numeric. 1A= 2 causes the computer to set A= 2 in line 1.

Exercise: Experiment with different types of characters permitted in a variable name:

```
T:  A= 1
R:  READY.
T:  AB= 2
R:  READY.
T:  ABC= 3
R:  READY.
T:  ABCABCABC= 4
R:  READY.
T:  A12345678= 5
R:  READY.
T:  A123456789123456789= 6
R:  READY.
```

Although the first character must be alphabetic, following characters may be either alphabetic or numeric, and the variable may be of any length that fits.

Regardless of the length of the variable, only the first two characters are used to distinguish between variables.

Exercise: Verify that only the first two characters are used in a variable name.

```
T:  PRINT A1
R:  6
R:  READY.
```

Notice that the variables A123456789123456789, A12345678, and A1 all refer to the same quantity. Verify that by printing out all three values.

```
T: PRINT A1234567891234567 89;  A12345678;  A1
```

(NOTE: Although we show the above on one line, the VIC screen can only handle 23 characters to a line and will divide the PRINT statement accordingly. Keep this in mind as you continue.)

```
R:  6    6    6

R:  READY.
```

The first two characters are the only ones used, but additional characters can be used to describe what the variable stands for.

```
T:  ABCNT=33
R:  READY.
T:  PRINT AB
R:  33
R:  READY.
```

ABCNT could stand for the count of values A and B.

II. CBM BASIC CALCULATOR MODE

Calculator mode means that the statements entered will be executed immediately. These statements have no line numbers. The absence of a line number signals the software that this statement is to be executed immediately and not entered into the program memory. Calculator mode was used to determine the legal variable names in the preceding examples.

When typing lines, remember: each line must end with the 'RETURN' key pressed, T: tells you what to type, and R: tells you the response you should get on the screen.

```
T:  A=5
R:  READY.
T:  B=10
R:  READY.
T:  C=A*B
R:  READY.
```

These three lines had no visible effect on your VIC. However, the three variables A, B, and C were set to the values specified. To check that C was calculated properly, type:

```
T:  PRINT C
R:  50
R:  READY.
```

To check all three variables at once:

```
T:  PRINT A;B;C
R:  5  10  50
R:  READY.
```

Each two variables in the output list are separated by two spaces.

Calculator mode can be used to calculate values directly in the PRINT statement.

```
T:  PRINT  5*10
R:  50
R:  READY.
T:  PRINT A*B+C
R:  100
R:  READY.
```

A shorthand for the PRINT statement in the CBM BASIC is the question mark (?).

```
T:  ?25*10
R:  50
R:  READY.
```

A. Using Strings in CBM BASIC

```
T:  A$="NAME IS"
```

We will no longer show the computer's READY response; it will be assumed. Double quote is used to define the limits of a string of characters. Enter some character strings.

```
T:  PRINT A$
R:  NAME IS
T:  B$="MY"
T:  PRINT B$+A$
R:  MYNAME IS
```

The + (plus sign) concatenates the two strings A$ and B$ for the display only. Concatenate means to attach. No space is provided between MY and NAME. Let's put one in.

```
T:  PRINT B$+"b"+A$
R:  MY NAME IS
T:  B$="bMYb"
T:  PRINT B$+A$
R:   MY NAME IS
```

Be careful when you type $+. If you don't release the shift key before typing you will get a ?SYNTAX ERROR. The line will look OK since shifted + is the graphics symbol for "cross". To save the concatenated string:

```
T:  C$=B$+A$
T:  PRINT C$+"bDAVE"
R:   MY NAME IS DAVE
```

B. Numeric and Fractional Values

Type in a few numbers to get a feel for your VIC numeric capabilities.

```
T:  X=.25
T:  Y=10
T:  Z=X*Y
T:  PRINT Z
R:  2.5
T:  ? Y/X
R:  40
```

A ? can be substituted for the PRINT command. A / means division; means exponentiation.

```
T:  ? Y 3
R:  1000
T:  PRINT Z
R:  2.5
T:  PRINT INT(Z)
R:  2
```

INT takes the integer part of a number.

C. Conversion of Data

There are several functions available in CBM BASIC to convert from one form of data representation to another. You have already seen INT, which converts from a fractional number to an integer.

```
T: PRINT ASC("Z")
R: 90
```

ASC converts a character to its equivalent number; CHR$ converts a number to its equivalent character.

```
T:  L=ASC("Z")
T:  PRINT L; CHR$(L)
R:  90  Z
T:  L=ASC("ZXCV")
T:  ?L
R:  90
```

The number 90 corresponds to the character Z. ASC only converts the first character of the string argument.

```
T:  PRINT CHR$(256)
R:  ?ILLEGAL QUANTITY
R:   ERROR
```

The legal characters are 0-255.

```
T:  PRINT CHR$(255)
R:  
```

The number 255 corresponds to the character .

Exercise: What is the integer value of "?"? What is the character for 71?

```
T:
R:
T:
R:
```

D. Balance Your Checkbook in Calculator Mode

Your VIC can be used to help you balance your checkbook. Let the variable OB stand for the old balance.

```
T:  OB=990.25
T:  OB=OB - 12.36
T:  ?OB
R:  977.89
```

$977.89 is the balance after subtracting a $12.36 check.

```
T:  OB=OB - 10.00 - 20.00 - 22.50 - 11.98
T:  ?OB
R:  913.41
```

This is the balance after subtracting four more checks. Let's try that again typing only those digits that are required.

```
T:  OB=977.89
T:  OB=OB - 10 - 20 - 22.5 - 11.98
T:  PRINT OB
R:  913.41
```

This is the same as the previous example except that unneeded digits were not typed.

E. Reserved Words

CBM BASIC reserves the use of certain strings of characters to specify operations to be performed.

Exercise: Determine what the VIC does with improperly used reserved words.

```
T:  INT=3
R:  ?SYNTAX
R:   ERROR
```

Since INT means the integer function, it cannot be used as a variable. What would you expect from

```
T:  ASC=1
R:
R:
T:  CHR$=1
R:
R:
T:  PRINT=12
R:
R:
```

F. Modes of Variables and Constants

There are two modes for variables and constants: character strings and floating point. Mixing of modes is not permissable between character strings and numeric values.

Exercise: Mixing strings and numeric data is not permitted.

```
T:  A=CHR$(63)
R:  ?TYPE MISMATCH
R:   ERROR
```

The error message occurred because A is a numeric variable and CHR$ is a character function.

Exercise: Display the numerical equivalent of the character "?".

```
T:  A=ASC("?")
T:  PRINT A
R:  63
```

What will you get with

```
T:  D$=CHR$(63)
T:  PRINT D$
R:
```

III. INPUTTING A PROGRAM

Every program line begins with a line number. However, any statements entered without line numbers are executed immediately. That's the calculator mode you used in Chapter II. Any line of text typed to BASIC that begins with a digit is processed only after a RUN command. There are four possible actions that may occur:

1. A new line is added to the program. This occurs if the line number is legal (range 0-63999 for the VIC).

2. An existing line is modified. This occurs if the line number matches the line number of an existing line in the program. That line is modified to have the text of the newly typed line.

Exercise: Use the line editor to change a statement in your BASIC program.

```
T:  100 A=B+C
```

If you want to change "B" to "D" in line 100, one way is to

```
T:  100 A=D+C
```

3. An existing line is deleted when a number is typed that is equal to an existing line number. The line must contain only the line number; otherwise, the line will be replaced and not deleted.

4. An error message (?SYNTAX ERROR) is generated if the line number is out of range (0-63999).

A. Blanks

Blanks preceding a line number are ignored. The first nondigit in a line terminates the line number. Multiple blanks are permitted anywhere in a line for readability, but not in reserved words (PRINT, for instance) or constants. Since BASIC ignores spaces between variables, operators and keywords in statements (except within quotes), it doesn't really matter whether you type PRINT A ; B or PRINTA;B. We have attempted to be consistent with spacings in this workbook, but you will notice some variation.

B. Multiple Program Statements

Multiple program statements may appear on a single line if separated by a colon (:). A line number must appear only at the beginning of the first statement on the line. A memory saver is the elimination of the ending quote (") on strings, since the " requires one byte of memory. For example, PRINT "ABC works just as well as PRINT "ABC" as long as the character string is the last part of the line. However, ending quotes must be used in the following case: 100 PRINT "A=",X To avoid ending up with programs that are hard to read and debug, use matching quotes even though it takes an additional byte.

Some examples of multiple statement lines:

```
T:  NEW
T:  10 PRINT"AAA:PRINT"BBB:PRINT"CCC
T:  RUN
R:  AAA:PRINT   0
R:  CCC
```

The system prints AAA:PRINT as a character string since it is contained between two quotes. It then thinks that BBB is a variable and prints the value of BBB, which is 0. PRINT "CCC is the next statement and it is handled correctly. The above example shows some of the problems encountered when a quote is left out.

Exercise: The proper use of the quote in multiple statement lines.

```
T:  10 PRINT"AAA":PRINT"BBB":PRINT"CCC"
T:  RUN
R:  AAA
R:  BBB
R:  CCC

T:  10 A$="AB"
T:  20 PRINT A$
T:  RUN
R:  AB
```

Exercise: Multiple statements per line are also permitted in calulator mode.

```
T:  FOR I=1 TO 20:PRINT I:NEXT I
R:                  (The numbers 1 through 20 will be displayed)
```

C. Typing Mistakes

If a typing mistake occurs during the entering of a line, three ways to correct the error are:

1. Hit 'DEL' until the proper number of characters are deleted. Retype the rest of the line from that point.

2. Hit 'RETURN' and retype the entire line.

3. Hit 'CLR', which clears the screen and doesn't enter the line into memory.

D. CBM BASIC Commands

LIST line number - line number.

The LIST command is used to display your program (see Fig. 3). The display will be in ascending order by line numbers regardless of the order in which the statements were originally entered. A blank is automatically inserted between the line number and the first non-numeric character of the line, even if a blank was not originally entered.

If a LIST command is given without a line number, all statements in memory are displayed.

If a LIST 100 is given, then line 100 only will be displayed.

If a LIST 100-150 is given, all the statements from 100 through 150 will be displayed.

If a LIST 100- is given, line 100 and all those following will be displayed.

If a LIST -100 is given, all lines up to and including 100 will be displayed.

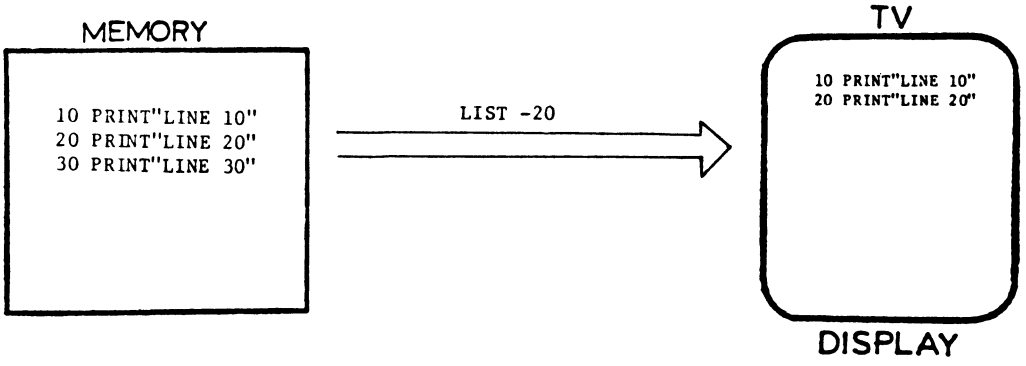
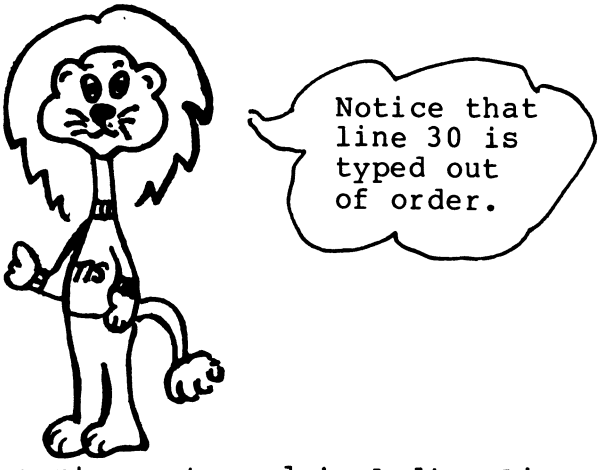


Fig. 3. LIST Command Action.

Exercise: Use of the various options on a LIST command.

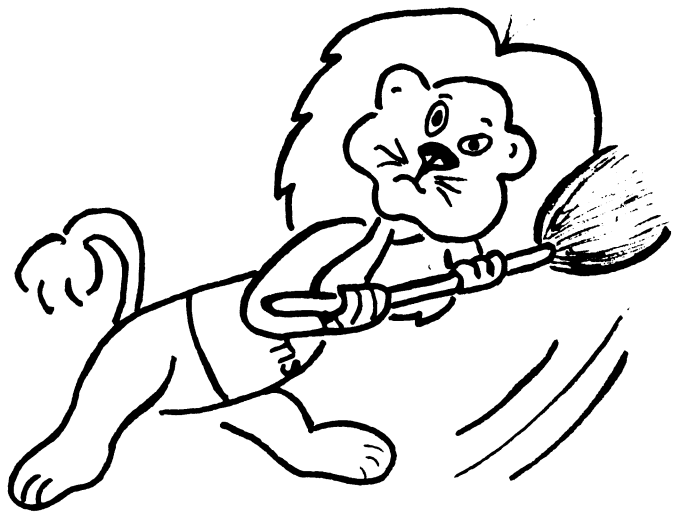
```
T:  NEW
T:  10 PRINT"LINE 10"
T:  30 PRINT"LINE 30"
T:  20 PRINT"LINE 20"
T:  LIST
R:  10 PRINT"LINE 10"
R:  20 PRINT"LINE 20"
R:  30 PRINT"LINE 30"
T:  LIST -20
R:  10 PRINT"LINE 10"
R:  20 PRINT"LINE 20"
T:  LIST 20-
R:  20 PRINT"LINE 20"
R:  30 PRINT"LINE 30"
```



LIST -20 lists all statements up to and including line 20.

LIST 20- lists all statements from line 20 on.

NEW



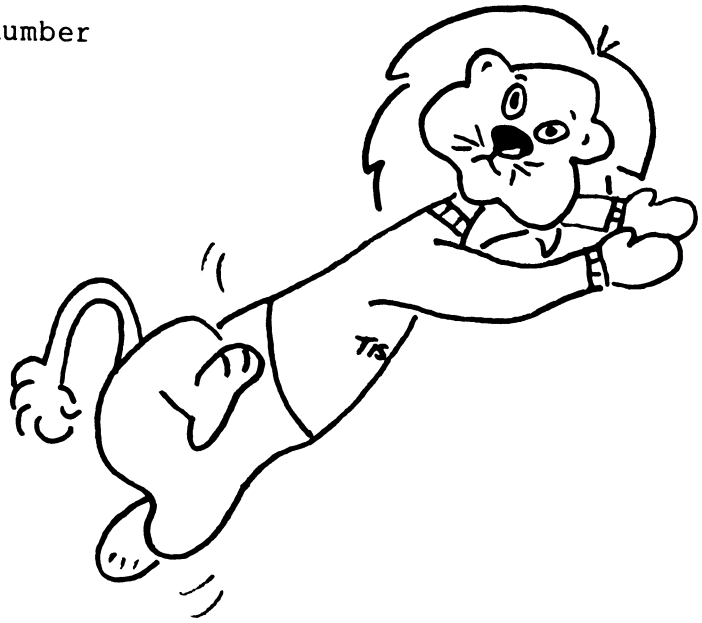
NEW is used to reset memory before entering a new program. If a NEW command is not given and you begin to enter another program, the system cannot tell the difference between the new program and modification of the old program. If you want to keep the old program in memory, be sure to start the new program at a line number greater than the largest line number of the old program.

Exercise: Show that NEW removes a program from memory.

```
T:  NEW
T:  10 REM NEW TEST
T:  LIST
R:  10 REM NEW TEST
T:  NEW
T:  LIST
R:  READY.
```

Trying to display (LIST) the sample program after typing NEW shows that the program is gone.

RUN line number



The line number is optional; if left out, program execution will start with the lowest line number and continue until all program statements have been executed or until an END or STOP statement is encountered, whichever occurs first. If line number is given, execution begins with that line and proceeds to execute from there.

Exercise: Control of execution of a program with RUN.

```
T:  NEW
T:  10 PRINT"LINE 10"
T:  20 PRINT"LINE 20"
T:  30 PRINT"LINE 30"
T:  40 END
T:  50 PRINT"LINE 50"
T:  60 PRINT"LINE 60"
T:  RUN
R:  LINE 10
R:  LINE 20
R:  LINE 30
```

RUN starts execution at the lowest line number and continues until an END or STOP is encountered or there are no more statements.

```
T:  RUN 50
R:  LINE 50
R:  LINE 60
```

RUN 50 begins execution with statement 50 and continues until there are no more statements.

```
T:  RUN 20
R:  LINE 20
R:  LINE 30
```

RUN 20 begins execution with statement 20 and stops when the END statement is encountered.

Exercise: Demonstration of two programs in memory at the same time.

```
T:  NEW
T:  10 R=3
T:  20 A=
T:  30 PRINT A
T:  40 END
```

This first program (10-40) calculates the area of a circle with radius 3

```
T:  100 X=
T:  110 A=SIN(X)
T:  120 PRINT A
T:  130 END
```

The second program (lines 100-130) calculates the sign of a 90 degree angle. There are now two separate programs in memory and we will soon show you how to run each one.

Exercise: Selective execution of one of two programs in memory. The programs from the previous exercise must still be in memory.

```
T:  RUN
R:  28.27343339
```

This programs displays the area of a circle whose radius was typed in line 10.

E. 'STOP' and CONT

When a program is in an endless loop, pressing the 'STOP' key will cause the program to stop execution and will display:

BREAK IN statement number

The statement number will depend on when 'STOP' is typed. Typing CONT will resume execution where it left off; that is, with the next statement.

Exercise: Use of the 'STOP' and the CONT command. Input the following program.

```
T:  NEW
T:  10 I=0
T:  20 FOR A=1 TO 500
T:  30 X=A
T:  40 NEXT A
T:  44 I=I+1
T:  46 PRINT I
T:  50 GO TO 20
T:  RUN
R:  1
R:  2
 :
 :
T:  'STOP'
R:  BREAK IN 30    This line number depends on when 'STOP' is
T:  CONT                                                typed.
R:
```

If your program is waiting for input, 'STOP' will not give you a BREAK. While waiting for input 'RETURN' followed by 'STOP' will cause the program to stop.

IV. GETTING INFORMATION OUT OF YOUR PROGRAM

A program can calculate many different values. Unless you have some way to see the results, those calculations are useless. See Fig. 4 for how to display information calculated. Results can be displayed on the TV screen in several different ways.

A. Output Formats - Numeric Data

Exercise: Output of uninitialized variables in field format. Uninitialized variables are ones you haven't set yet. BASIC automatically sets those to zero.

```
T:  NEW
T:  PRINT A,B,C,D,E
R:   0              0
R:   0              0
R:   0
```

Numeric fields are each 11 character positions wide and start in display columns as follows. (If the first column is numbered "1"):

FIELD	COLUMN
1	1
2	12
3	1 of second line
4	12 of second line
5	1 of third line

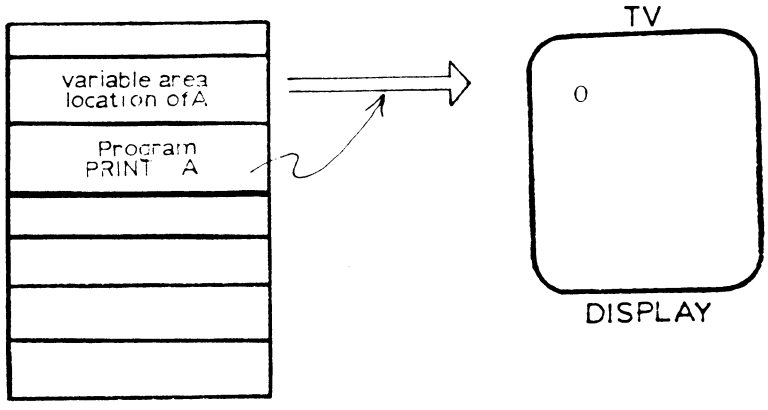


Fig. 4 PRINT action.

Exercise: Show the way different types of numbers are displayed. (The colon is used to put multiple statements on one line.)

```
T:  NEW
T:  100  A=-1:  B=-2
T:  110  PRINT A,B
T:  120  A=1:  B=2
T:  130  PRINT A,B
T:  140  A=-111:  B=-222
T:  150  PRINT A,B
T:  160  A=111:  B=222
T:  170  PRINT A,B
T:  RUN
R:  -1              -2
R:   1               2
R:  -111            -111
R:   111             111
```

Notice that one space is reserved at the beginning of each field for the number's sign (+ or -). However, with positive numbers, the sign is not printed; thus positive numbers are actually printed starting in columns 2 and 13.

Exercise: Test printing several items with comma spacing.

```
T:  PRINT 1,2,3,4,5,6,7,8,9,10,11,12
R:   1               2
R:   3               4
R:   5               6
R:   7               8
R:   9               10
R:   11              12
```

Exercise: Output of initialized variables in continuous format.

```
T:  A=1:B=2:C=3
T:  PRINT A;B;C
R:   1  2  3
```

When semicolons are used to separate elements of a print list, each positive numeric item is separated from the next by two blank spaces. The first item is preceded by a space.

Exercise: Show the way different types of numbers are displayed in semicolon format.

```
T:  NEW
T:  100 A= -1 : B= -2 : C= -3
T:  110 PRINT A;B;C
T:  120 A=1 : B=2 : C=3
T:  130 PRINT A;B;C
T:  140 A= -111 : B= -222 : C= -333
T:  150 PRINT A;B;C
T:  160 A=111 : B=222 : C=333
T:  170 PRINT A;B;C
T:  RUN
T:  -1  -2  -3
T:   1   2   3
T:  -111  -222  -333
T:   111   222   333
```

Notice that as with comma spacing, a space is reserved at the beginning of the number for the sign. In addition, one space is printed following each number. Thus negative numbers are separated by one space, and positive numbers by two (one space for separation, one for the sign).

Exercise: Display various values to demonstrate output conventions.

```
T:  NEW
T:  100  A=1
T:  110  FOR I=1  TO 4
T:  120  A=A*10
T:  130  PRINT A,A
T:  140  PRINT A;A
T:  150  PRINT
T:  160  NEXT I
T:  RUN
```

Notice how the numbers are displayed.

B. Output Formats - Character Strings

Exercise: String output in field and continuous formats.

```
T:  NEW
T:  100  A$="A":B$="B":C$="C":D$="D"
T:  105  E$="E":F$="F":G$="G"
T:  110  PRINT A$,B$,C$,D$,E$,F$,G$
T:  120  PRINT A$;B$;C$;D$;E$;F$;G$
T:  RUN
R:  A               B
R:  C               D
R:  E               F
R:  G
R:  ABCDEFG
```

A comma in the PRINT statement puts the strings in successive fields (11 columns apart). Notice that there is no reserving of a space for a sign, as with numeric variables.

A semicolon in the PRINT statement results in strings being printed together, with no separating spaces.

To summarize the display output (PRINT) conventions for both numeric and string ; reserves one space for a numerical sign, which remains blank if the sign is positive. In addition, numbers are separated by one space. Strings are printed together, with no spaces. , uses 11 - character fields. Strings are printed at the beginning of each field; with numbers, the first space in the field is reserved for the number's sign.

C. Spacing

Exercise: Use the SPC function to display blanks.

```
T:  NEW
T:  200 FOR I=0 TO 22
T:  210 PRINT SPC(I);"A"
T:  220 NEXT I
T:  RUN
R:  (A diagonal of A's will be displayed.)
```

The SPC function allows you to print a variable number of spaces before displaying the data specified. A diagonal of A's will be displayed. If the display moves too rapidly for you to check where each A is placed, hold down the 'CTRL' key. That will slow the display.

Exercise: Use the TAB function to space data on the display.

```
T: 210 PRINT TAB(I);"A"
T: RUN
```

The TAB function will print beginning in the column defined by the variable (I). The difference between TAB and SPC is that TAB defines the column to start printing in while SPC defines the number of spaces between fields. Notice that the first column is column 0, not 1!

Exercise: Show the difference between TAB and SPC.

```
T:  NEW
T:  10 FOR I=2 TO 10 STEP 2
T:  20 PRINT SPC(I);"A";SPC(I);"B"
T:  30 PRINT TAB(I);"C";TAB(I+2);"D"
T:  40 PRINT
T:  50 NEXT I
T:  RUN
R:    A  B
R:    C D
R:       A     B
R:       C D
      .
      .
      .
R:               A        B
R:               C D
```

Line 20 results in I spaces (different in each cycle) being printed between A and B. Line 30 gives a constant single space between C and D.

V. GETTING INFORMATION INTO YOUR PROGRAM

A. Design Goals

One design goal should be to minimize the amount and type of information the person running the program must enter. The data required should be short, simple, and logical. The more input required, the longer it will take and the more likely an error will occur.

In writing a program, a good rule of thumb is to never use a constant where there is even the remotest possibility of a change being needed later. In those cases, use a variable instead of a constant.

On the surface, these two philosophies appear to conflict. However, BASIC provides several different methods of getting data into your program, that is, setting variables to particular values.

The key words for getting information to your program are:

INPUT
READ and DATA

B. INPUT

INPUT allows the user of the program to provide data for the program to process. This data can be either control information to select options and specify how the processing is to be done, data to be processed, or both. A message may also be displayed at the same time input is asked for by enclosing the message in quotes following INPUT.

Exercise: Show how to prompt for input.

```
T:  NEW
T:  10 INPUT "WHAT IS YOUR NUMBER"; N
T:  20 ? N
T:  30 IF N<> 0 THEN 10
T:  RUN
R:  WHAT IS YOUR NUMBER?
T:  5
R:  WHAT IS YOUR NUMBER?
T:  0
R:  READY.
```

Prompts must be short. If the prompt is longer than one line of 23 characters, the prompt will be treated as if you typed it as data.

Exercise: Demonstrate the restrictions on prompts.

```
T:  NEW
T:  100 INPUT "THIS IS A VE
    RY LONG PROMPT";I
T:  110 PRINT I
T:  RUN
R:  THIS IS A VERY LONG PR
    OMPT?"
T:  5
R:  ?REDO FROM START
R:  THIS IS A VERY LONG PR
    OMPT?
T:  'STOP' 'RESTORE'
```

Let's see what INPUT thought you entered.

```
T:  100 INPUT "THIS IS A VERY LONG PROMPT";I$
T:  110 PRINT I$
T:  RUN
R:  THIS IS A VERY LONG PR
    OMPT?
T:  5
R:  THIS IS A VERY LONG PR
    OMPT?5
R:  READY.
```

This shows that INPUT thought you typed the prompt in, as well as the 5. The solution is to keep your prompt shorter than a line.

INPUT is used to read data from the VIC keyboard (see Fig. 5). When the INPUT statement is executed, a ? prompt is given on the TV display.

1. Floating-Point Input . Floating-point input allows you to enter very large or very small numbers into your Challenger.

Exercise: A program to explore legal floating-point input data.

```
T:  NEW
T:  4000 INPUT V:PRINT V:GO TO 4000
T:  RUN
R:  ?
```

Since V is a floating-point variable, we can use this fragment of a program to experiment with the format and size of numbers that the CBM BASIC will accept. The ? implies that the program is ready for INPUT.

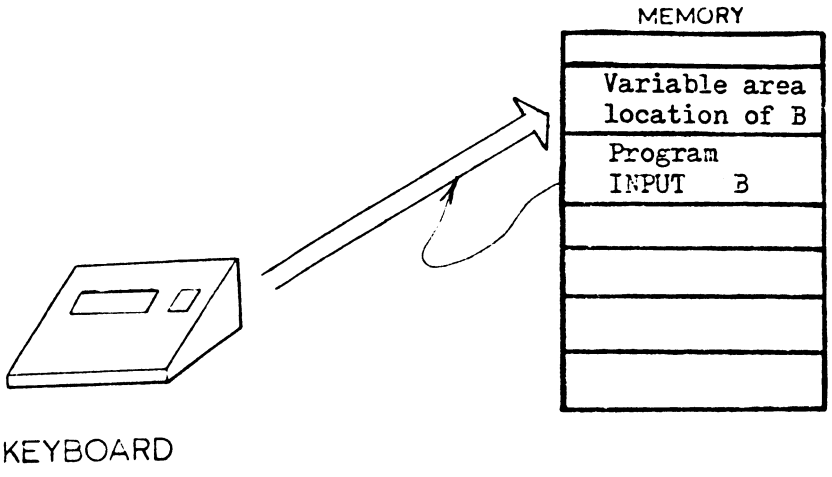


FIG. 5 INPUT action

```
T:  1.5
R:  1.5
R:  ?
```

Normal fractional numbers can be typed.

```
T:  1.5E+5
R:  150000
R:  ?
```

Scientific notation can be used for large numbers.

```
T:  .005E+6
R:  5000
R:  ?
T:  1235.01E+2
R:  123501
```

Scientific notation can be used to move the decimal point (scale) a number.

```
R:  ?
T:  12.5E+40
R:  ?OVERFLOW
R:   ERROR IN 4000
```

This message means that 12.5E+40 is too large. Earlier we saw that 1.5E+5 was not too large. What is the largest number that the Challenger can handle? See Section VI, Data Representation, for a small program to determine experimentally the largest and smallest numbers that the VIC can represent.

The overflow error caused the program to terminate. To continue putting data in you must:

```
T:  RUN
R:  ?
```

To check the effect of leading blanks, type:

```
T:  bbb5
R:   5
R:  ?
```

Try several combinations of leading, embedded, or trailing blanks to determine how they are handled by the VIC.

```
T:

R:

R:  ?

T:

R:

R:  ?

T:

R:

R:  ?

T: 'RETURN' only.

R:

R:  ?
```

When 'RETURN' only is typed the previous value is used. To terminate this program type 'RETURN' and 'STOP' simultaneously. It may take a few tries to get BREAK IN 4000. Another way to terminate this program is to type 'STOP' and 'RESTORE' at the same time. This method cleans the screen and changes back to the standard colors.

Remember that if you find a combination that generates an error message and the READY response, you must type:

```
T:  RUN
```

to continue with your input testing.

Some characters are not permitted for INPUT to a floating-point variable.

```
T:  RUN
R:  ?
T:  5 + 6
R:  ? REDO FROM START
R:  ?
```

This error message indicates unacceptable (bad) input. It does not terminate the program. It does give you another chance. What would you expect to get if you:

```
T:  BCD
R:
R:  ?
```

Since the INPUT statement was expecting a floating-point number, a letter (like B) would be considered unacceptable.

Try a few other symbols to determine what is legal.

```
T:

R:

R:  ?

T:

R:

R:  ?

T:

R:

R:  ?
```

If you didn't try something like

```
T:  123,456
```

Try it now.

The response:

```
R:  ? EXTRA IGNORED
R:   123
R:  ?
```

means that you provided more items of data than the INPUT statement requested. The comma is a data item separator.

```
T:  'RETURN'  'STOP' (simultaneously or 'STOP' 'RESTORE')
R:  OK
```

2. String Input . A little earlier we tried unsuccessfully to enter some non-numeric characters. The error was caused because the INPUT variable was floating-point.

INPUT is used to input strings or numeric values from the keyboard. INPUT works essentially the same when inputting numeric values or strings.

For example:

```
T:  NEW
T:  10 INPUT A$,A
T:  20 PRINT A$,A
T:  RUN
R:  ?
```

At this point you may input a string of characters, a comma, and a numeric value. The total number of characters entered must be less than 72. The comma separates the two input values.

```
T:  ATEST=,10
R:  ATEST=                    10
```

In the previous example you can put in the string you want, then hit 'RETURN'. The system will then prompt with another ? and wait until you input the numeric value. each value may be entered on a separate line by hitting a 'RETURN' for each entry of on the same line by using commas between each value.

Exercise: Use the above program to input values a line at a time.

```
T:  RUN
R:  ?
T:  ATEST=
R:  ??
T:  10
R:  ATEST=                    10
```

Exercise: A program to explore legal string input data.

```
T:  NEW
T:  2000 INPUT X$:PRINTX$:GO TO  2000
T:  RUN
R:  ?
T:  ASDE
R:  ASDE
R:  ?
```

Alphabetic characters are legal input for string (X$) variables but not for floating-point.

```
T:  "ASWER"
R:  ASWER
R:  ?
```

Note that the leading and trailing quotes are stripped off, that is, not included in the string.

```
T:  TYPE""QUOTE
R:  TYPE""QUOTE
R:  ?
```

Quotes are only removed if they are at the beginning of the line.

```
T:  bb"AFR"
R:  AFR
```

Leading blanks are removed and quotes stripped off. Remember, b means a blank or space.

```
T:  bbbFGH
R:  FGH
R:  ?
T:  'RETURN' 'STOP' (simultaneously or 'STOP' 'RESTORE')
R:  READY.
```

Leading blanks are removed. Modify your test program to see the treatment of trailing blanks.

```
T:  2000 INPUTX$:Z$=X$+X$:PRINTZ$:GO TO 2000
T:  RUN
R:  ?
T:  XC
R:  XCXC
R:  ?
```

With no blanks specified, Z$=X$+X$ simply concatenates the two copies of the input string.

```
T:  XCbb
R:  XCXC
R:  ?
```

Trailing blanks are not retained.

```
T:  bbbABCbbb
R:  ABC   ABC
R:  ?
```

Both leading and trailng blanks are removed.

```
T:  bbXYZbbABbb
R:  XYZ  AB  XYZ  AB
R:  ?
```

Embedded blanks are permitted.

```
T:  "bbbABCbbb"
R:     ABC      ABC
R:  ?
```

All blanks are included when enclosed in quotes.

```
T:  "bA
R:   A A
R:  ?
```

A single quote will preserve leading blanks.

```
T:  Ab"
R:  A "A "
R:  ?
T:  "Abb
R:  AA
R:  ?
T:  ABC,
R:  ?EXTRA IGNORED
R:  ABCABC
R:  ?
```

A comma is treated as a data separator for string variables just as it is for numeric variables.

```
T:  "ABC,
R:  ABC,ABC,
R:  ?
```

A quote permits the comma to be part of the string and not considered as a separator.

Try several examples of using special characters in a string both with and without quotes.

```
T:

R:

R:  ?

T:

R:

R:  ?

T:  'RETURN' 'STOP'
```

Many published programs use encoded responses to select options. With string handling there is no need to make encoded responses, such as 1 for yes, 0 for no. You can make your programs easier to use by people with little computer background by accepting words for input.

Exercise: Program to allow human-oriented input to programs for option selection.

```
T:  NEW
T:  1500 INPUT"YES, NO, OR MAYBE";Z$
T:  1510 IF Z$="YES" THEN PRINT "YEP"
T:  1520 IF  Z$="NO"  THEN  PRINT"NOPE"
T:  1530 IF Z$="MAYBE" THEN PRINT "MAKE UP YOUR MIND"
T:  1540 IF Z$=""THEN STOP
T:  1550 GO TO 1500
T:  RUN
R:  YES, NO, OR MAYBE?
```

Be sure to use a semicolon (;) between the last quote and the variable Z$ in line 1500. Remember prompt must be less than one line.

If you get a "syntax error" message, recheck the line number specified in the error message by:

```
T:  LIST line number
```

The most likely errors are the wrong number of quotes, a misspelled keyword, an omitted $, or a comma where a semicolon should be.

Test this program by trying:

```
T:  YES
R:

R:  YES, NO, OR MAYBE?
T:  NO
R:

R:  YES, NO, OR MAYBE?
T:  MAYBE
R:

R:  YES, NO OR MAYBE?
T:  ILLEGAL

R:  YES, NO, OR MAYBE?
T:  'RETURN'
```

C. READ, DATA, and RESTORE

The READ statement is used to allow the programmer to easily change the value of variables (parameters) in the program. READ requires a matching DATA statement in the program.

Data may be put in a program by the DATA statement and a corresponding READ statement. A DATA statement must have a corresponding value for each variable read. However, a RESTORE will restore the DATA statements so that they may be read more than once.

Exercise: Use DATA, READ, and RESTORE statements to get strings into your program.

```
T:  NEW
T:  10 DATA "A","B"
T:  20 DATA C,D
T:  30 READ A$,B$
T:  40 PRINT A$,B$
T:  50 READ C$,D$
T:  60 PRINT C$,D$
T:  70 RESTORE
T:  80 GOTO 30
```

```
T:  RUN
R:  A                B
R:  C                D
R:  A                B
R:  C                D
     .
     .
     .
     .
```

The program is now in a continuous loop because RESTORE allows the DATA statements to be read over and over.

```
T:  'STOP'
R:  BREAK IN 40  (This statement number depends on when the
    'STOP' key is hit.)
T:  70
T:  RUN
R:  A     B
R:  C     D
R:  ?OUT OF DATA
R:   ERROR IN 30
```

When the RESTORE is removed (statement 70) the DATA statements are only "read" once. Notice that in a DATA statement a string character does not have to be enclosed in quotes. See statement 20.

VI. DATA REPRESENTATION

A. Largest Numeric Value

To find the largest floating-point number that the VIC can handle, run the following program.

Exercise: Test for the floating-point maximum.

```
T:  NEW
T:  200 I=1
T:  210 I=I+I
T:  220 PRINT I
T:  230 GO TO 210
T:  RUN
```

A long series of numbers starting with

```
R:  2
R:  4
R:  8
R:  16
```

will be printed out; the last number will be

```
R:  4.25352959E+37
R:  8.50705917
```

followed by

```
R:  ?OVERFLOW
R:   ERROR IN 210
```

This means that 1.701411835E+38 was too large to represent.

```
T:  ?1.701411834E+38                (? is shorthand for PRINT)
R:   1.701411835E+38
T:  ?1.701412E+38
R:  ?OVERFLOW
R:   ERROR
```

From this you can see that 1.701411834E+38 is the largest number that the VIC can handle.

Exercise: Modify the program above to display large negative floating-point numbers to determine the largest negative number that the VIC can represent. If the sign of the result alternates between plus and minus, you should try a different modification. Your result should show that the sign is independent of the maximum size permitted.

```
T:
T:
R:
R:
```

The largest negative number is _____?

B. Smallest Numeric Value

To find the smallest floating-point number that the VIC can handle, run the following program:

Exercise: Test for the floating-point minimum.

```
T:  NEW
T:  100 I=1
T:  110 I=I/2
T:  120 PRINT I
T:  130 IF I <> 0 THEN 110
T:  RUN
```

A long series of numbers starting with

```
R:  .5
R:  .25
R:  .125
R:  .
R:  .
R:  .
```

will be printed out; the last two will be

```
R:  2.93873588E-39
R:  0
```

This means that 1.46936794E-39 was too small and had to be represented as a zero.

```
T:  ?2.93873588E-39
R:   2.93873588E-39
T:  ?2.93873587E-39
R:   0
```

From this you can see that 2.93873587E-39 is the smallest number that the VIC can handle. Anything smaller is converted to zero.

Exercise: Modify the program above to display small, negative floating-point numbers to determine the smallest negative number that the VIC can represent.

```
T:
T:
R:
R:
```

The smallest negative number that the VIC can represent is _____?

C. Integer Range

To find the largest integer number that the VIC can handle.

```
T:  NEW
T:  300 I%=0
T:  305 I%=I%+1
T:  310 J%=2 I%-1
T:  320 PRINT J%
T:  330 GO TO 305
T:  RUN
```

A series of numbers

```
T:  1
T:  3
T:  7
T:  .
    .
    .
R:  32767
R:  ?ILLEGAL QUANTITY
R:   ERROR IN 30
```

will be printed out. This means that 65535 is too large to represent as an integer.

```
T:  I%=32768
R:
```

This means that 32767 is the largest number that can be represented as an integer.

Exercise: Modify the program above to display large negative integers.

```
T:
T:
R:
T:
R:
```

D. Memory Space Used

To determine the memory space required for data storage, we will use the FRE function. FRE returns the amount of space remaining.

These are the responses for a 5K machine. If you have more memory, note the values.

Exercise: Determine the amount of space used by a single floating-point value.

First, determine the amount of memory space left after reserving 100 floating-point locations.

```
T:  NEW
T:  DIMA(100)
T:  ? FRE(0)
R:  3069
```

Next, determine the amount of space left after reserving 101 floating-point locations.

```
T:  NEW
T:  DIMA(101)
T:  ? FRE(0)
R:  3064
```

The difference (5) is the number of bytes one floating-point values requires. Now determine how much space an integer value takes.

```
T:  NEW
T:  DIMA%(100)
T:  ?FRE(0)
R:  3372
T:  NEW
T:  DIMA%(101)
T:  ?FRE(0)
R:  3370
```

The difference (2) is the number of bytes one integer value takes.

Exercise: Determine the amount of space used by a single string value.

```
T:  NEW
T:  DIM A$(100)
T:  ? FRE(0)
R:  3271
T:  NEW
T:  DIM A$(101)
T:  ? FRE(0)
R:  3268
```

For one additional string variable, three bytes are required. However, each string can have as many as 255 characters in it.

In addition to the space used for each value, 7 bytes are used for each variable name and pointers.

E. Number of Significant Digits

The VIC can display six digits in a number even though more information can be stored in memory.

Exercise: Determine the number of digits displayed for floating-point numbers.

```
T:  PRINT 123456789
R:  123456789
```

This indicates that nine digits can be displayed.

```
T:  PRINT 1234567890
R:  1.23456789E+09
```

This shows that the tenth digit is not displayed.

Exercise: Show how many digits of information can be saved in memory.

```
T:  A=1234567890:PRINTA
R:  1.23456789E+09
T:  B=1234567891:PRINTB
R:  1.23456789E+09
```

Even though the numbers entered differed by 1 in the tenth digit, the display shows that they are equal. Are they really?

Exercise: Test if the numbers entered are really equal.

```
T:  IF A<>B THEN PRINT "<>"
R:  <>
```

The response shows that the VIC can distinguish the difference of 1 in the tenth digit even if it doesn't display the difference. Can the VIC distinguish a difference of 1 in the eleventh digit?

```
T:  C=12345678900:PRINT C
R:  1.23456789E+10
T:  D=12345678901:PRINT D
R:  1.23456789E+10
```

The displayed values appear to be equal. Are they?

```
T:  IF C<>D THEN PRINT "<>"
R:
```

It appears that the VIC can not distinquish between two numbers that differ by one in the eleventh position.

The number of significant digits that are displayed is nine, and the number saved in memory is about ten.

F. Rounding

Since the VIC saves about ten digits in memory and only displays nine digits, it must "round" to display the proper value.

Exercise: Determine how the VIC rounds numbers for display.

```
T:  NEW
T:  100 A=1234567890
T:  110 FOR I=1 TO 15
T:  120 B=A+I
T:  130 PRINT "A+";I;"=";B
T:  140 NEXT I
T:  RUN
R:  A+ 1 = 1.23456789E+09
 .
 .
 .
R:  A+ 5 = 1.2345679E+09
 .
 .
 .
R:  A+ 15 = 1.23456791E+01
```

From this display you can see that 1234567895 was rounded up to 1.2345679E+09 and 1234567905 was rounded up to 1.23456791E+09. The VIC rounds the ninth digit up whenever the tenth digit is 5 (or more) and rounds down whenever the tenth digit is 4 (or less).

VII. USING THE CASSETTE FOR PROGRAM STORAGE

The basic VIC does not have any way to save program or data. Each time you turn the basic VIC off all information is lost and must be reentered the next time you turn the VIC on. For this reason, one of your first additions will most likely be a mass storage device. You have three choices: a tape cassette, a floppy diskette or both. Since the cassette is substantially cheaper and currently more readily available, program storage will be described in terms of the cassette. If you are fortunate enough to have diskette simply add ,8 on the end of each described command.

Example:

```
SAVE "FILENAME"     SAVE "FILENAME",8
VERIFY "FILENAME"   VERIFY "FILENAME",8
LOAD "FILENAME"     LOAD "FILENAME",8
```

The response you get when using the diskette will be slightly different than from the cassette, but the end result will be the same.

Now onward with the cassette storage of programs.

A. SAVE a Program

Since typing is so much work, you will want to save programs that you have entered. The command to do that is SAVE "LABEL".

Exercise: Write a program on the cassette. Put a scratch (one with no useful information on it) tape into the tape recorder.

```
T:  10 REM SAVE PROGRAM TEST
T:  SAVE
R:  PRESS RECORD & PLAY ON
R:  TAPE
R:  OK - This occurs after you press play and record.
R:  SAVING
R:  READY.
```

If you wish to give your program a label (name), type SAVE"LABEL". "LABEL" may be from 1-83 characters.

Exercise: Write a program on the tape with the label ABC. Rewind cassette.

```
R:  SAVE"ABC"
R:  PRESS RECORD & PLAY ON
R:  TAPE
R:  OK - This occurs after you press play and record.
R:  WRITING ABC
R:  READY.
```

WARNING!WARNING! The system does not check to see which buttons are pushed on the cassette. If any button is left on, the system assumes it is the right one. Whatever button you push is assumed to be the correct one.

B. VERIFY a Program

In order to be confident that the program you wrote on the cassette is written correctly, you may compare what is on the tape with what is in memory with the VERIFY command. First, rewind the cassette.

Exercise: Verify that the information just written is correct.

```
T:  VERIFY
R:  PRESS PLAY ON TAPE
R:  OK
R:  SEARCHING
R:  FOUND
R:  VERIFYING
R:  OK
R:  READY
```

If there is a compare error, you will get a ?VERIFY ERROR. If this happens, rewind the tape and try again. Since cassette reading and writing is not all that reliable, trying again often gets past the problem.

If a VERIFY is typed without a label, the system will try to compare the first program it finds on the cassette with what is in memory. If a VERIFY "LABEL" is typed, the system will search for the program with the given label. The name of each program encountered will be listed.

Cassettes may be removed without rewinding, then replaced, and a new program written on the cassette from that point. A 10-15 second leader is written after each SAVE command is given.

Exercise: SAVE a program and demonstrate that VERIFY catches differences. First, rewind a blank cassette.

```
T:  NEW
T:  10 PRINT"TAPE TEST"
T:  20 INPUT L,W
T:  30 A=L*W
T:  40 PRINT A,L,W
T:  50 END
T:  SAVE"A"
```

```
R:  PRESS RECORD & PLAY ON
R:  TAPE
R:  OK
R:  SAVING
R:  READY.
```

This puts a small program into memory and saves it on cassette with the name "A". Now rewind the cassette.

```
T:  VERIFY
R:  PRESS PLAY ON TAPE
R:  OK
R:  SEARCHING
R:  FOUND A
R:  VERIFYING
R:  OK
R:  READY.
```

You have succeeded in checking the information on cassette against the program in memory. Do <u>not</u> rewind the cassette. Just press stop on the cassette.

```
T:  SAVE"ABC"
R:  PRESS RECORD & PLAY ON
R:  TAPE
R:  OK
R:  SAVING ABC
R:  READY.
```

This saved a second copy of our program on the cassette. The second copy is after the first and is called "ABC".

Rewind the cassette.

To force a verify error, we will change the program in memory.

```
T:  10 PRINT"FORCE ERROR"
T:  VERIFY"ABC"
R:  PRESS PLAY ON TAPE
R:  OK
R:  SEARCHING FOR ABC
R:  FOUND A
R:  FOUND ABC
R:  VERIFYING
R:  ?VERIFY
R:   ERROR
R:  READY.
```

C. LOAD a Program

Before you load a program into memory let's check what is there right now.

```
T:  LIST 10
R:  10 PRINT"FORCE ERROR"
```

To reload your original program rewind the cassette and type

```
T:  LOAD
R:  PRESS PLAY ON TAPE
R:  OK
```

The LOAD command automatically clears memory and reads into memory the first program on the cassette. The followng responses are displayed after you press play on the recorder.

```
R:  SEARCHING
R:  FOUND A
R:  LOADING
R:  READY.
R:  LIST 10
R:  10 PRINT "TAPE TEST"
```

If you want to load a labeled program, rewind the cassette and type LOAD"ABC".

Exercise: LOADing of programs SAVEd previously.

```
T:  LOAD"ABC"
R:  PRESS PLAY ON TAPE
R:  SEARCHING FOR ABC
R:  FOUND A
R:  FOUND ABC
R:  LOADING ABC
R:  READY.
```

If you have more than one program on a cassette and a LOAD is typed, the first program found is the one loaded. For example, if you have three programs on a cassette and the recorder is sitting at the end of the first program, LOAD will load the second program.

If a LOAD"ABC" is given, a search will be made for program ABC. When ABC is found it will be loaded. A list of all programs found before ABC will be displayed. For example, <u>suppose</u> there are four programs on a cassette, the first two are labeled 1 and $, respectively; the third is not labeled; and the fourth is labeled VIC.

If a LOAD "VIC" is given, the system responds as follows:

```
T:  LOAD"VIC"
R:  PRESS PLAY ON TAPE
R:  OK
R:  SEARCHING FOR VIC
R:  FOUND 1
R:  ROUND $
R:  FOUND
R:  FOUND VIC
R:  LOADING
R:  READY.
```

If you do not know what is on a cassette, just ask to LOAD a nonexistent label and each program on the cassette will be listed as it is found.

WARNING!WARNING! Loading a program from tape is not the most reliable operation in the world. The most common error we have run into is that the system finds the label it is searching for but never successfully loads the program. The only way to correct this is to rewind the cassette and try again. Often, a second or third try will be successful.

VIII. BRANCHING

The VIC BASIC has two statements for branching. The GO TO (or GOTO) statement is an unconditional one-way branch to a single specified statement number. The ON X GO TO statement is an unconditional multi-way branch to one of several statement numbers. The particular statement chosen is specified by X and the list of possible statement numbers. In some cases, the space between GO and TO will cause a syntax error. Because of this always use GOTO with no space.

A. GO TO

An unconditional branch (transfer of control) to a single statement is written as GO TO nnn where nnn is a statement number. The statement number must be a numeric constant. Variables or string constants are not allowed.

Exercise: Check for legal statement numbers in the GO TO statement.

```
T:  NEW
T:  100 PRINT "GO TO 100":END
T:  GOTO 100
R:  GOTO 100

T:  GOTO "100"
R:  ?UNDEF'D STATEMENT
R:   ERROR

T:  S=100:GOTO S
R:  ?UNDEF'D STATEMENT
R:   ERROR

T:  S$="100:GOTO S$
R:  ?UNDEF'D STATEMENT
R:   ERROR
```

You can see that a numeric constant is the only legal way to specify the destination of a GO TO.

The statement number that you want to GO TO must exist or an error message will be given.

```
T:  GOTO 110
R:  ?UNDEF'D STATEMENT
R:   ERROR
```

Even though GObTO is acceptable in these examples, it is a good idea to always type GOTO (with no blank). Since, in some cases, the blank is <u>not</u> allowed.

B. ON X GO TO

The multi-way branch is written ON X GO TO nl, n2,...,nn where X is the variable that selects which branch to take and the n's are the possible statements to branch to. The destination is the xth n. For example, if X is 2, then control is transferred to the 2nd statement number in the list (n2). If X is 4, control is transferred to the 4th statement in the list (n4).

Exercise: Show how ON X GOTO works.

```
T:  NEW
T:  1 PRINT "STATEMENT #1":STOP
T:  2 PRINT "STATEMENT #2":STOP
T:  3 PRINT "STATEMENT #3":STOP
T:  4 PRINT "STATEMENT #4":STOP
T:  5 PRINT "STATEMENT #5":STOP
T:  100 X=2
T:  110 ON X GOTO 1,2,3,5
T:  120 PRINT "SLIPPED BY GOTO"
T:  RUN 100
R:  STATEMENT #2
R:  BREAK IN  2
```

If you get STATEMENT #1 and BREAK IN 1, you probably typed just RUN and not RUN 100. Try RUN 100. If you get ? SYNTAX ERROR IN 110 remove the space between GO and TO.

```
T:  100 X=4
T:  RUN 100
R:  STATEMENT #5
R:  BREAK IN 5
```

Notice that the ON X went to the 4th statement number in the list and NOT to statement #4.

Exercise: Show how ON X handles values outside the legal range.

```
T:  100 X=0
T:  RUN 100
R:  SLIPPED BY GO TO

T:  100 X=5
T:  RUN 100

R:  SLIPPED BY GO TO
```

If the variable is outside the quantity of statement numbers provided, the next statement in sequence is executed.

Exercise: Show the legal range of values for the variable X in ON X...

```
T:  100 X=255
R:  RUN 100
R:  SLIPPED BY GOTO

T:  100 X=256
T:  RUN 100
R:  ?ILLEGAL QUANTITY
R:   ERROR IN 110

T:  100 X =-1
T:  RUN 100
R:  ?ILLEGAL QUANTITY
R:   ERROR IN 110
```

This shows that the selected value must be between 0 and 255.

Exercise: Use ON X GO TO to select one of four actions. Let's say that you are writing a program to prepare a mailing list and that there are four actions to take.

1. Add a new name to the list.
2. Change an existing name.
3. Print out the list.
4. Count the number of entries on the list.

You could set up the program to accept as input the numbers 1-4 to select the action.

```
T:  NEW
T:  100 PRINT "MAILING LIST SKELETON"
T:  110 INPUT "ENTER ACTION #";A
T:  130 ON A GOTO 1000, 2000, 3000, 4000
```

The choice of statement numbers is somewhat arbitrary. You can make your programs more readable by developing a consistent pattern (style). It is easy to see that for A=n the statement to go to is n000.

```
T:  150 GOTO 110
T:  1000 PRINT "ADD SELECTED"
T:  1990 GOTO 150
T:  2000 PRINT "CHANGE SELECTED"
T:  2990 GOTO 150
T:  3000 PRINT "PRINT LIST REQUESTED"
T:  3990 GOTO 150
T:  4000 PRINT "COUNT REQUESTED"
T:  4990 GOTO 150
```

For each possible action number a stub is provided. The stub just tells you whether the control portion of the program is working correctly or not. Later, the PRINT statements can be replaced with the program steps to do the task described.

It may seem strange to have statement #150 (the GOTO 110) and each stub end with a GOTO 150. A "GOTO" another "GOTO" sounds very inefficient. It is inefficient. A GOTO directly is a little faster than a "GOTO" another "GOTO." However, if you use this approach of always returning to the statement after the ON X GOTO statement, you won't have to guess where the program is going. That makes debugging much easier.

Now try all the legal inputs.

```
T:  RUN
R:  MAILING LIST SKELETON
R:  ENTER ACTION #?

T:  1
R:  ADD SELECTED
R:  ENTER ACTION #?

T:  2
R:  CHANGE SELECTED
R:  ENTER ACTION #?

T:  3
R:  PRINT LIST REQUESTED
R:  ENTER ACTION #?

T:  4

R:  COUNT REQUESTED
R:  ENTER ACTION #?
```

To terminate this program, type 'RETURN' and 'STOP' (simultaneously). Call this program MAIL1 and save it on cassette. If you have forgotten how to save a program on cassette see Section VII. If you do not have a way to save the program you will have to reenter it later. We will use MAIL1 later and add features to it.

Exercise: Check how MAIL1 reacts to errors in input. If you have turned power off, reload MAIL1. To reload you need to:

```
T:  LOAD
R:  PRESS PLAY ON TAPE
R:  OK
R:  SEARCHING
R:  FOUND MAIL1
R:  LOADING
```

Now you are ready to test the program.

```
T:  RUN
R:  MAILING LIST SKELETON
R:  ENTER ACTION #?
T:  0
R:  ENTER ACTION #?
T:  5
R:  ENTER ACTION #?
```

The program does not handle bad input very well.

You can use the fact that some out-of-range numbers GO TO the next statement to improve error handling. To get out of the program, you can simply press 'RETURN' and 'STOP' or 'STOP' 'RESTORE'.

```
T:  140 PRINT "INPUT MUST BE 1-4"

T:  RUN
R:  MAILING LIST SKELETON
R:  ENTER ACTION #?
T:  0
R:  INPUT MUST BE 1-4
R:  ENTER ACTION #?
```

Save this version of the program as MAIL2. Later, we will use it as a base to make a more human-oriented interface. Since the VIC can handle character strings as well as numbers, there is no reason that MAIL can't respond to requests specified by characters. In the section on IF, another improvement will be made to handle negative and large numbers for the value of A.

IX. THE IF STATEMENT

The IF statement allows a program to do different things based on input parameters or the current value of a calculated value.

A. THEN Form

The most general form of the IF statement is: IF expression THEN statement. When the "expression" is true, that is, evaluates to non-zero, the "statement" is executed next. If the "expression" is false, that is, evaluates to zero, the "statement" is skipped and the following numbered line is executed next.

Exercise: Use the THEN form of the IF statement.

```
T:  NEW
T:  100 A=1
T:  110 IF A=1 THEN PRINT "A=1"
T:  120 PRINT "LINE 120"

T:  RUN
R:  A=1
R:  LINE 120

T:  100 A=2
T:  RUN
R:  LINE 120
```

B. GO TO Form

The IF statement does not have to have a THEN. The form: IF expression GOTO nn is legal; it is the same as: IF expression THEN GOTO nn. A third way to specify the same operattion is: IF expression THEN nn.

All three forms give the same result. If the expression is true, continue execution at line nn. If the expression is false, continue execution at the next numbered line.

Exercise: Try the GOTO form of the IF statement. Use part of the program already entered.

```
T:  130 STOP
T:  140 PRINT "A=1"
T:  150 STOP
T:  110 IF A=1 GOTO 140
T:  RUN
```

```
R:  LINE 120
R:  BREAK IN 130
```

Since A=2, the IF test failed. This caused statements 120 and 130 to be executed.

```
T:  100 A=1
T:  RUN
R:  A=1
R:  BREAK IN 150
```

Here A=1 and statements 140 and 150 were executed.

Exercise: Use the short form of the GOTO in an IF statement.

```
T:  110 OF A=1 THEN 140
T:  RUN
R:  A=1
R:  BREAK IN 150

T:  100 A=2
T:  RUN
R:  LINE 120
R:  BREAK IN 130
```

These two runs show the same results as the previous exercise.

C. Multiple Statements

The THEN form of the IF statement can have multiple statements after the THEN. These multiple statements are executed only if the test is true.

Exercise: Show how multiple statements after the THEN are handled.

```
T:  NEW
T:  100 A-1
T:  110 IF A=1 THEN PRINT"A=1":A=2:PRINT"A IS NOW=";A
T:  120 PRINT "LINE 120"
T:  130 STOP
R:  RUN
R:  A=1
R:  A IS NOW=2
R:  LINE 120
R:  BREAK IN 130
```

When the IF test is true, all the statements after the THEN are executed.

```
T:  110 IF A=1 THEN PRINT"A=1":GOTO 130:PRINT"NEVER HERE!"
T:  RUN
R:  A=1
R:  BREAK IN 130
```

The statement after GOTO 130 can never be executed. The GOTO prevents the PRINT "NEVER HERE!" from being reached in sequence. Since that PRINT does not have a statement (line) number, it cannot be reached by another GOTO. Since only one line number is permitted per line, you cannot insert a line number before the PRINT.

Exercise: Show that the second line number on a line cannot be reached.

```
T:  110 IF A=1 THEN PRINT"A=1":GO TO 130:115 PRINT"NEVER HERE!"
T:  RUN
R:  A=1
R:  BREAK IN 130

T:  GO TO 115
R:  ?UNDEF'D STATEMENT
R:   ERROR
```

This shows that statement 115 cannot be found. Only the regular statement number on a line can be reached. Let's remove the GOTO 130 and see what happens with two statement numbers on one line.

```
T:  110 IF A=1 THEN PRINT"A=1":115 PRINT"NEVER HERE!"
T:  RUN
R:  A=1
R:  ?SYNTAX
R:   ERROR IN 110
```

You can use what you have learned about the IF statement to improve the MAIL2 program.

Exercise: Add error checking to MAIL2. Reload MAIL2. To refresh your memory on how this program handles erroneous input, try:

```
T:  RUN
R:  MAILING LIST SKELETON
R:  ENTER ACTION #?
T:  -1
R:  ?ILLEGAL QUANTITY
R:   ERROR IN 130

T:  RUN
R:  MAILILNG LIST SKELETON
R:  ENTER ACTION #?
T:  256
R:  ?ILLEGAL QUANTITY
R:   ERROR IN 130
```

The following two lines will catch bad input and give the program a chance to continue.

```
T:  120 IF A<0 THEN 140
T:  122 IF A>255 THEN 140
```

```
T:  RUN
R:  MAILING LIST SKELETON
R:  ENTER ACTION #?
T:  -5
R:  INPUT MUST BE 1-4
R:  ENTER ACTION #?
T:  256
R:  INPUT MUST BE 1-4
R:  ENTER ACTION #?
T:  'RETURN' 'STOP' or 'STOP' 'RESTORE'
```

The results show that the program no longer terminates when bad input is typed. Save this program as MAIL3.

Exercise: Make MAIL3 more human oriented. Change the input to accept the strings "ADD", "CHANGE", "PRINT", and "COUNT" instead of the numbers 1 to 4. Load MAIL3. A complete listing of the character-oriented mailing list skeleton is in Appendix A.

```
T:  101 AT$(1)="ADD"
T:  102 AT$(2)="CHANGE"
T:  103 AT$(3)="PRINT"
T:  104 AT$(4)="COUNT"
```

This sets up the array AT$ with the four action words. You could put any string you desired in here.

```
T:  110  INPUT "ENTER ACTION",A$
T:  112 IF A$="" THEN STOP
T:  115 FOR A=1TO4
T:  120 IF A$<>AT$(A) THEN 135
T:  122
T:  135 NEXT A
```

Get the action request and search for a match.

```
T:  140  PRINT "INPUT MUST BE LEGAL!"
```

Change error message to reflect what really is wrong.

```
T:  RUN
R:  MAILING LIST SKELETON
R:  ENTER ACTION?
T:  ADD
R:  ADD SELECTED
R:  ENTER ACTION?
```

Try a few legal and illegal command strings to convince yourself that the program works. Why don't you have to test explicitly for a value of A<0 or A>255?. Save this version of the mailing list skeleton as MAIL4.

X. *SUBROUTINES*

As you become more experienced in programming you will probably write bigger and more complicated programs. Subroutines can help you get these programs to work more quickly and easily. Subroutines are useful to avoid repeated sections of almost identical statements, to simplify the organization of the program, to make the program easiering to read and follow, and to make it easier to debug.

Once a subroutine has been debugged, you can use it as a building block in other programs. This is particularly useful if you can be sure that this building block does not conflict with or interfere with other building blocks.

To use a subroutine you must keep in mind two principles. The first is linkage. When you want to use a subroutine, your program has to call (invoke) it. When the subroutine is finished, program execution must return to where the subroutine was called. Figure 4-1 shows this linkage. Subroutines are most useful if they can be called from several different places in your program. For this reason, you should try to put into a subroutine a function that you need done frequently and in several different places.

The second principle of subroutines is parameter passing. Since BASIC has no method for passing parameters in the subroutine call, you will have to establish your own conventions. Parameters are data for the subroutine to work with or on. Parameters (called arguments in some circles) can be input (to the subroutine) or output (values returned to the main program by the subroutine).

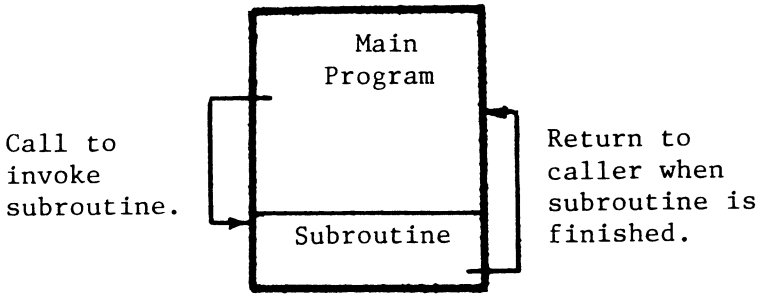


Fig. 6. Subroutine Linkage.

A. GO SUB

To get to a subroutine, the statement GOSUB nnn must be executed. The statement at line number nnn is executed next. GOSUB works the same as GOTO except that GOSUB keeps around information on where it (GOSUB) was called. This allows the program execution to return to where the subroutine was called.

B. RETURN

The RETURN statement is used to end a subroutine and get back to where the subroutine was called.

Exercise: Show how subroutines are used.

```
T:  NEW
T:  100 PRINT "MAIN PROGRAM"
T:  110 GOSUB 1000
T:  120 PRINT "MAIN PROGRAM AFTER SUB"
T:  130 STOP
T:  1000 PRINT "SUB 1000 CALLED"
T:  1190 RETURN

T:  RUN
R:  MAIN PROGRAM
R:  SUB 1000 CALLED
R:  MAIN PROGRAM AFTER SUB
R:  BREAK IN 130
```

Exercise: Show what happens if a RETURN is executed with no GOSUB.

```
T:  RUN 1000
R:  SUB 1000 CALLED
R:  ?RETURN WITHOUT GOSUB
R:   ERROR IN 1190

T:  110 GOTO 1000
T:  RUN
R:  MAIN PROGRAM
R:  SUB 1000 CALLED
R:  ?RETURN WITHOUT GOSUB
R:   ERROR IN 1190
```

Exercise: Check the maximum subroutine nest depth. That is, how many GOSUB's can you do without a RETURN?

```
T:  NEW
T:  5 I=-1
T:  10 I=I+1
T:  20 GOSUB 10
T:  RUN
```

```
R:  ?OUT OF MEMORY
R:   ERROR IN 10
T:  PRINT I
R:  23
```

The maximum depth of subroutine nesting that the VIC allows is 23.

C. ON X GO SUB

The GOSUB statement is similar to the GOTO statement in that a single destination (line number) is specified. Just as you can make a multiway unconditional branch with the ON X GOTO nl,n2,...you can select one of several subroutines with ON X GOSUB nl,n2,...

Exercise: Show how ON X GO SUB...works.

```
T:  NEW
T:  100 INPUT "ENTER I";I
T:  110 ON I GOSUB 1000, 2000, 3000
T:  120 PRINT "BACK TO MAIN"
T:  130 GOTO 100
T:  1000 PRINT "FIRST (1000) SUBROUTINE"
T:  1990 RETURN
T:  2000 PRINT "SECOND (2000) SUBROUTINE"
T:  2990 RETURN
T:  3000 PRINT "THIRD (3000) SUBROUTINE"
T:  3990 RETURN

T:  RUN
R:  ENTER I?
T:  1
R:  FIRST (1000) SUBROUTINE
R:  BACK TO MAIN
R:  ENTER I?

T:  0
T:  BACK TO MAIN
R:  ENTER I?
```

When 0 or number greater than 3 is entered ON X GOSUB works just like ON X GOTO. That is the next statement is performed. To terminate this program:

```
T:  1E40
R:  ?OVERFLOW
R:   ERROR IN 100
```

XI. STRINGS

Each computer system has its own character set, that is, a collection of symbols which the computer recognizes. These symbols may be digits, alphabetic letters, or special characters. In our case, we are talking about the symbols on the VIC keyboard. A number is assigned to each character (symbol) that represents the character in the computer. A string is defined to be one or more characters. A string may be a message, a name an address, etc.

To permit intercomputer communication and establish a standard, the American Standard Code for Information Interchange (ASCII) character set was defined. It includes all letters (both upper and lower case), digits, and many specials symbols. Some of the characters used in BASIC are taken from the ASCII character set. When we talk about strings in this book, we mean the ASCII character set and its numerical representation. For example, the ASCII character A is represented by the number 65, the character B by the number 66, and the character 1 by the number 49. The numerical representation of the keyboard uses the numbers 0-255.

The string functions make the VIC a powerful and versatile computer. By using the string functions the VIC can truly become a personal computer.

It is reasonably easy to tailor your programs so that they actually carry on a dialogue with you. ANIMAL and ELIZA are examples of programs that make extensive use of strings to carry on a dialogue. Strings are also useful in computer-aided instruction and in searching library indexes.

The best way to learn to use strings is to write programs that use strings. As you go through the examples in this section, be sure you understand what is being done before you continue. Some of the examples are easy to ready and understand; other examples are more complicated. All examples should be tried on your VIC.

A. Legal String Names

A string name is a variable name followed by a dollar sign ($). A string name may be up to 85 characters, including the dollar sign ($). Only the first two characters of the name are actually used, however. The first character must be alphabetic.

---

1 101 Games, David Ahl

2 Creative Computing July-Aug 1977 Vol 3, No 4

Since only the first two characters are used in the variable name, you must be careful in your naming conventions. A string is defined by enclosing it in quotes.

Exercise: Check string variable names.

```
T:  NEW
T:  ABCDE$="J"
T:  AB123$="K"
T:  PRINT ABCDE$,AB123$
R:  K           K
T:  PRINT AB$
R:  K
T:  A123456789$="S"
T:  PRINT A1$
R:  S
```

B. Subscripted String Variables

A subscripted variable need not be dimensioned if the subscript will not exceed 10. A non-subscripted variable is unique and distinct from a subscripted variable of the same name. For example, A$ is distinct from A$(0) or A$(1).

Exercise: Show the independence of variable names.

```
T:  NEW
T:  10 A$="A"
T:  20 A$(0)="B"
T:  30 A$(1)="C"
T:  40 A$(2)="D"
T:  50 PRINT A$;A$(0);A$(1);A$(2)
T:  RUN
R:  ABCD
```

Note that 0 is a valid subscript. Also note in statement 10, A$ is not subscripted and is unique and distinct from all subscripted variables of the same name. It was not necessary to DIMension A$(0)-A$(2) because the subscript does not exceed 10.

Exercise: Show that string arrays greater than 10 must be dimensioned.

```
T:  10 FOR I=1 TO 11
T:  20 A$(I)="*"
T:  30 NEXT I
T:  40 END
T:  RUN
R:  ?BAD SUBSCRIPT
R:   ERROR IN 20
```

In the above example, A$ must have a DIM A$(11) statement since the subscript value does exceed 10.

C. Maximum String Length

Exercise: Determine the maximum string length.

```
T:  NEW
T:  FOR I= 1 TO 255:A$=A$+"D":NEXT I 255
```

Note that 255 characters in a string is ok.

```
T:  NEW
T:  FOR I= 1 TO 256:A$=A$+"D":NEXT I
R:  ?STRING TOO LONG
R:   ERROR
```

Note that 256 characters in a string is invalid. The maximum permissible string length is 255 characters.

XII. OPERATIONS ON STRINGS

A. Statements

A string variable may be lengthened by concatenation. Concatenate means to attach.

Exercise: Concatenate three strings.

```
T:  NEW
T:  10 A$="V"
T:  20 B$="I"
T:  30 C$=A$+B+"C"
T:  40 PRINT C$
T:  RUN
R:  VIC
```

Let's look at the way the + sign is interpreted when used with numeric values and when used with string values.

Exercise: Show how the + sign is used with numbers and with characters.

```
T:  NEW
T:  10 A$="123"
T:  20 B$="456"
T:  30 A=123
T:  40 B=456
T:  50 C$=A$+B$
T:  60 C=A+B
T:  70 PRINT C$,C
T:  RUN
R:  123456          579
```

In statement 50, the string B$ is concatenated (attached) to the right-hand side of the string in A$. In statement 60, a numeric addition is made.

Strings need to be manipulated in a different manner than numeric values. The + sign is the only arithmetic operator that is valid to use with character strings. However, the relational operators >,<, and = sign do work with strings in the following manner. A character is less than, equal to, or greater than the numerical representation of the character it is being compared to. Since the numerical representation of B is 66 and A is 65, A is less than B. The Boolean (logical) operators AND, OR, and NOT are invalid when used with strings.

Exercise: Show that the logical operators are not allowed with strings.

```
T:  C$=A$ OR B$
R:  ?TYPE MISMATCH
R:   ERROR
```

B. Functions

1. ASC . The ASC function returns the numeric value corresponding to the ASCII character of the first character in a string. This function may be used to find the numeric values of the VIC character set.

   Exercise: Determine the numerical value of any VIC character.

```
T:  NEW
T:  10 FOR I=1 TO 1000
T:  20 INPUT A$
T:  25 IF A$="" THEN STOP
T:  30 A=ASC(A$)
T:  40 PRINT A$,A
T:  50 NEXT I
T:  RUN
R:  ?
T:  A                    65
T:  any character
R:
R:  'RETURN'
T:
R:  BREAK IN 25
```

2. CHR$. The CHR$ function returns a character corresponding to the integer given. The integer value must be between 0 and 255. The CHR$ function is the inverse of the ASC function, that is, it converts in the opposite direction. The CHR$ function is given an integer value and returns the corresponding character. The ASC function is given a character and returns the corresponding integer value.

   Exercise: Display the numerical equivalent of each character.

```
T:  NEW
T:  10 FOR I=0 TO 255
T:  20 B$=CHR$(I)
T:  30 PRINT I,B$
T:  40 GET A$:IF A$="" THEN 40
T:  50 NEXT I
T:  RUN
R:  0
T:  any key
```

This program will display an integer code (0-255) and the character corresponding to that integer (some integers will not display a character on the

screen in this code) each time you type any key on the keyboard.

To terminate this program type 'STOP' 'RESTORE'. To restore upper case

```
T:  ?CHR$(142)
```

Exercise: Show that CHR$ is the inverse of ASC. Use the previous program as a base.

```
T:  25 A=ASC(B$)
T:  30 PRINT A;B$;I
T:  RUN
R:  0          0
T:  any key
    .
    .
    .
```

Statement 20 converts the value of I to its equivalent VIC character. Statement 30 converts the character back to its numeric value. Statement 50 allows you to control the rate at which values are displayed.

Exercise: Demonstrate the type of CHR$.

```
T:  ?CHR$(142)
T:  NEW
T:  10 B=CHR$(5)
T:  20 PRINT B
T:  RUN
R:  ?TYPE MISMATCH
R:   ERROR IN 10
```

Remember that CHR$ returns a character and that B is a real variable. To help remember the type of variable to use, remember ASC has no ($) dollar sign, therefore, the variable name must not have a ($) sign. Since CHR$ has a dollar sign ($) the variable name must also have a dollar sign ($).

```
B$ = CHR$(40)
 C= ASC("B")
```

3. <u>LEFT$</u> . The LEFT$ function is used to select characters from the left-hand portion of a stirng. To use LEFT$, give the string name and a count of the characters chosen from the left side of the string.

Exercise: Select characters from a string.

```
T:  NEW
T:  10 A$="ABCDE"
T:  20 B$=LEFT$(A$,1)
T:  40 PRINT B$
T:  RUN
```

```
R:  A
R:  READY.
T:  20 FOR I = 0 TO 6
T:  30 B$=LEFT$(A$,I)
T:  50 NEXT I
T:  RUN
R:
R:  A
R:  AB
R:  ABC
R:  ABCD
R:  ABCDE
R:  ABCDE
R:  OK
```

Caution: No error message is given if I is greater than 5, that is, greater than the number of characters in A$. The full string is returned. Notice that when when I was 0 no characters were returned to B$.

4. RIGHT$. The RIGHT$ function is used to select a character from the right-hand portion of a string. It works just like LEFT$ except it selects characters starting from the right-hand side. To demonstrate this, just change LEFT$ to RIGHT$ in previous example.

   Caution: No error message is given if you exceed the length of the string. See LEFT$.

5. MID$. The MID$ function is used to select any character or group of characters from a string. MID$ is the most powerful function of the three functions LEFT$, RIGHT$ and MID$. To use MID$, just give the string name, the number of the character you want to start with, and the number (count) of characters you want to select. MID$ with only two arguments gives the rest of the string.

   Exercise: Select characters from the left of a string. Use the program from the previous exercise.

```
T:  30 B$=MID$(A$,1,I)
T:  RUN
R:
R:  A
R:  AB
R:  ABC
R:  ABCD
R:  ABCDE
R:  ABCDE
R:  READY
```

Now let's select characters starting from the right.

```
T:  5 J=0
T:  20 FOR I = 6 TO 0 STEP -1
T:  25 J=J+1
T:  30 B$=MID$(A$,I,J)
T:  RUN
R:  E
R:  DE
R:  CDE
R:  BCDE
R:  ABCDE
R:  ?ILLEGAL QUANTITY
R:   ERROR IN 30
R:  READY.
```

Caution: No error message is given if I or J or both are greater than the number of characters in A$.

6. LEN. The LEN function counts the number of characters in a string and returns that number.

Exercise: Use the LEN function to split a string into two strings of equal or almost equal length.

```
T:  NEW
T:  10 INPUT A$
T:  20 A=LEN(A$)
T:  30 B=INT(A/2)
T:  35 IF A=1 GOTO 50
T:  40 B$=LEFT$(A$,A-B)
T:  50 C$=RIGHT$(A$,B)
T:  60 PRINT B$,C$
T:  RUN
R:  ?
```

Type a string of your choice.

7. STR$. The STR$ function is used to convert a number to its string equivalent. If the number has more than 9 digits, rounding takes place and the value is converted to exponential form.

Exercise: Convert a number to its string equivalent.

```
T:  A=1234567885123
T:  PRINT STR$(A)
R:  1.23456789E+12
```

STR$ could be used to line up the decimal points in a checkbook balancing program.

Exercise: Write a program to line up the decimal points of a collection of numbers.

```
T:  NEW
T:  10 DATA 10.03,100.25,.25,6.13,0
T:  20 READ A$
T:  30 IF A$="0" GOTO 120
T:  40 FOR I=1 TO LEN(A$)
T:  50 IF MID$(A$,I,1)="." GOTO 70
T:  60 NEXT I
T:  70 FOR J=1 TO 10-I
T:  80 A$="b"+A$
T:  90 NEXT J
T: 100 PRINT A$
T: 110 GOTO 20
T: 120 END
T: RUN
R:    10.03
R:   100.25
R:      .25
R:     6.13
```

8. VAL . The VAL function is used to convert a string containing numeric characers to a numeric value. This function is the inverse of the STR$ function. If there are characters, other than digits in the string variable, only those digits to the left of the first non-numeric character will be converted to a numeric value.

Exercise: Convert a string to its numerical equivalent.

```
T:  NEW
T:  10 A$="1234"
T:  20 B$="A1234"
T:  30 C$="123B4"
T:  40 A=VAL(A$)
T:  50 B=VAL(B$)
T:  60 C=VAL(C$)
T:  70 PRINT A;B;C
T:  RUN
R:  1234      0       123
```

Variable A$ is converted as you would expect. Variable B$ is set to zero since the first character is a non-digit. Variable C$ is converted up to the first non-digit.

XIII. DATA REPRESENTATION AND PROCESSING PROGRAMS

If you are not already familiar with relation operators, number systems, character codes, logical operators, and how information is stored in memory, see Appendix B.

A. Number System Conversion Programs

The first topic we need to cover is conversion from one number system to another. The following exercises develope the needed programs.

Exercise: Write a program to display the binary representation of a decimal number. A complete listing of DEC-BIN appears in Appendix A.

```
T:  NEW
T:  90 REM DISPLAY BINARY VALUE
T:  100 INPUT "ENTER DECIMAL NO.";R
T:  430 R$=""
T:  440 GOSUB 2000
T:  500 PRINT "DECIMAL";R; "IS"
T:  510 PRINT "BIN ";R$
T:  520 PRINT
T:  530 GO TO 100
```

The main program prompts for and gets the decimal number. The subroutine at 2000 does the actual work. The main program then displays the result. The main program passes the number to convert to the subroutine in the variable R and receives the converted result in the variable R$.

```
T:  2000 REM CONVERT NO. TO BINARY CHARACTER STRING
T:  2010 C$(0)="0":C$(1)="1"
T:  2020 R2=R
T:  2030 T2=INT(R2/2)
T:  2040 S=R2-T2*2
```

S is the remainder, and T2 is the quotient.

```
T:  2050 R$=C$(S)+R$
T:  2060 R2=T2
T:  2065 IF LEN(R$)>16 THEN R$=RIGHT$(R$,16): RETURN
T:  2070 IF T2<>0 THEN 2030
```

The old quotient (T2) becomes the new dividend (R2). Repeat until all binary digits have been assembled in R$.

```
T:  2080 RETURN
```

Save this program as DEC-BIN. It will be used later to show how logic operations are performed.

```
T:  RUN
R:  ENTER DECIMAL NO?
T:  38
R:  DECIMAL 38 IS
R:  BIN 100110
R:  ENTER DECIMAL NO?
```

Try a few decimal numbers. You can use this program to check if you are able to successfully convert from decimal to binary by hand.

```
T:
R:
R:  ENTER DECIMAL NO?
T:
R:
```

To terminate this program type 1E40.

Exercise: Convert a binary number in a character string form to a decimal number. Load the DEC-BIN program if it isn't already in memory. We will use part of it. A complete listing of BIN-DEC appears in Appendix A.

```
T:  90 REM DISPLAY DECIMAL VALUE
T:  100 PRINT "ENTER BINARY NO"
T:  105 INPUT N$
T:  106 IF N$="" THEN STOP
T:  110 T1$=N$
T:  120 GOSUB1100
T:  130 IFT<0THEN100
T:  140 PRINTN$;"^BIN"
T:  145 PRINT "IS";T;"DECIMAL"
T:  150 GOTO100
```

The main program gets the character string with the binary number and passes it to the conversion subroutine (at 1100). If the conversion is successful, the value is printed out.

```
T:  1100 REM CONVERT BIN. CHAR. STRING TO DEC. NO.
T:  1105 T=0
T:  1110 L1=LEN(T1$)
T:  1120 FORI1=1TOL1
T:  1125 T=T*2
T:  1130 IFMID$(T1$,I1,1)="1"THENT=T+1:GOTO1150
T:  1140 IFMID$(T1$,I1,1)<>"0"THENT=-1:RETURN
T:  1150 NEXTI1
T:  1160 RETURN
```

The subroutine converts the binary character string to a decimal number. If any character besides 0 or 1 is found, an error flag is returned.

```
T:  RUN
R:  ENTER BINARY NO
R:  ?
T:  101
R:  101 BINARY IS
R:  5 DECIMAL
```

Try a few binary numbers. To end this program type "". Save this program as BIN-DEC.

B. Logical Operations Program

The three logical functions that are available in CBM BASIC are

a) AND
b) OR
c) NOT

If you are not familiar with these terms, see Appendix B. Each of these functions except NOT has two input values but only one output value (result). The NOT function has one input value and one output value. These values are normally referred to as "true" and "false." The VIC uses non-zero for true and zero for false.

Exercise: Display the value of the logical function performed on various data. Load the program BIN-DEC and add the following lines (a complete listing appears in Appendix A):

```
T:  90 REM  DISPLAY LOGICAL FUNCTIONS
T:  140 NE=T
T:  145
T:  150 PRINT "ENTER LOGICAL FUNCTION"
T:  160 INPUT "(A,O,N)";OP$
T:  190 F$=LEFT$(OP$,1)
T:  200 IF F$<>"A" AND F$<>"O" AND F$<>"N" THEN 150
```

Get the logical operator (AND, OR, NOT) and check if the first character is correct.

```
T:  210 TW$=""
T:  220 IF F$ = "N" THEN 420
```

Since NOT needs only 1 argument, skip past input of and conversion of second argument.

```
T:  300 PRINT"ENTER SECOND BINARY NO"
T:  305 INPUT TW$
T:  310 T1$=TW$
T:  320 GOSUB1100
T:  330 IF T<0 THEN 300
```

Make sure the number entered is a legal binary value.

```
T:  340 TWO=T
T:  400 IF F$="A" THEN R=NE AND TWO
T:  410 IF F$="O" THEN R=NE OR TWO
T:  420 IF F$="N" THEN R=NOT NE
```

Perform the requested logical operation on the input parameter(s).

```
T:  500 IF F$="N" THEN PRINT OP$;"b";N$;"bIS"
T:  505 IF F$<>"N" THEN PRINT N$;"b";OP$;"b";TW$;"bIS"
T:  510 PRINT R$
```

Print the input and operation with the result.

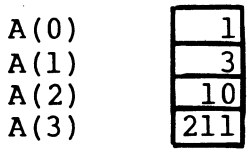
Save this program on a cassette. Call it LOGIC. Then try some test data.

```
T:  RUN
R:  ENTER BINARY NUMBER
R:  ?
T:  101
R:  ENTER LOGICAL FUNCTION
R:  (A,O,N)?
T:  A
R:  ENTER SECOND BINARY NO
T:  11
R:  101 A 11 IS
R:  1
R:  ENTER BINARY NO.
R:  ?
```

Try a few more values and operations to check the program. End the program by typing "" in response to request for first binary number.

XIV. SUBSCRIPTED VARIABLES

Just like a regular variable, a subscripted variable names a logical location in the computer's memory. These locations may be thought of as small, contiguous boxes, such as post office boxes, where the subscripted variable is the name of a box and the contents of the box is the value of the subscripted variable. Another way of thinking of these memory locations, is as rows or columns. In the example below, the memory locations are the boxes, the contents of the memory are the numbers in the boxes, and the address or name of the memory location is the subscripted variables on the left.



The value of A(0) is 1
The value of A(3) is 211
The name of the location of 3 is A(1)

An array of one column or one row is called a one-dimensional array or vector. Two-dimensional arrays consist of both rows and columns, sometimes called a matrix. The subscripts define the row and the column.

	Col 1	Col 2	Col 3
Row 1	A(1,1)	A(1,2)	A(1,3)
Row 2	A(2,1)	A(2,2)	A(2,3)

Subscripted variables make keeping track of ck of elements in an array or table much easier. In the VIC BASIC, numeric and string variables may be subscripted. Using subscripted variables make it easy to assign a lot of values to a lot of variables. The reason this is so easy is that subscripted variables work so well in FOR-NEXT loops. We will be using several examples where the subscripted variable is set to the value of the subscript.

Exercise: Input and output of non-subscripted and subscripted arrays.

```
T:  NEW
T:  10 DATA 5,7,8,11,12
T:  20 READ A1,A2,A3,A4,A5
T:  30 PRINT A1;A2;A3;A4;A5
R:  RUN
R:  5  7  8  11  12
```

Without subscripted arrays you must write each variable separately. This isn't bad with only five variables but think of 100. The following code uses subscripts to do the same thing.

```
T:  20 FOR I = 1 TO 5
T:  30 READ A(I)
T:  40 PRINT A(I);
T:  50 NEXT I
T:  RUN
R:  5  7  8  11  12
```

A. Legal Subscripts

Legal values for subscripts are non-negative integers. Variables, formulas, functions, and constants may be used for subscripts.

Exercise: Show that variables, functions, and formulas are legal subscripts.

```
T:  NEW
T:  10 A=2
T:  20 B=3
T: 30 C=5
T: 40 FOR I=1 TO 10
T: 50 D(I)=I
T: 60 NEXT I
T: 70 PRINT D(A*B-C)
T: 80 PRINT D(INT(C/A))
T: 90 PRINT D(C-INT(B/A)+A)
T:  RUN
R:  1
R:  2
R:  6
R:  OK
```

B. Dimensions

A subscripted variable does not need a corresponding DIM statement as long as the value of the subscript does not exceed 10 (this is true even in multiple subscripts). If you intend to use a subscript of 0, the maximum value of a subscript is still 10. So if you use 0 as a subscript you may have up to 11 items for each subscripted variable without needing a DIM statement for that variable.

Exercise: Find the limit of non-dimensioned subscripted numeric variables.

```
T:  NEW
T:  10 FOR J=1 TO 20
T:  20 FOR I=1 TO 20
T:  30 A(I,J)=I
T:  40 PRINT A(I,J);I;J
T:  50 NEXT I
T:  60 NEXT J
T:  70 END
T:  RUN
```

```
R:  1 1 1
R:  2 2 1
          .
          .
          .
R:  10  10  1
R:  ?BAD SUBSCRIPT
R:   ERROR IN 30
```

Without a DIM statementment for A the system will not continue when either I or J exceeds 10.

Exercise: Demonstrate a non-dimensioned subscripted variable starting with 0.

```
T:  NEW
T:  10 FOR I=0 TO 10
T:  20 A(I)=I
T:  30 PRINT A(I);I
T:  40 NEXT I
T:  RUN
T:  0 0
T:  1 1
T:  2 2
        .
        .
        .

R:  10 10
R:  READY.
```

The following list is the absolute maximum allowable DIMensions for an 5K VIC without getting a ?OUT OF MEMORY ERROR. These were checked out using the calculator mode, which means there was no program used in the machine. Be sure to type NEW before each test.

```
DIM A(713)          DIM A(25,26)          DIM A(8,8,7)
DIM A%(1785)        DIM A%(41,44)         DIM A%(11,11,11)
DIM A$(1190)        DIM A$(34,33)         DIM A$(10,9,9)
```

You can combine subscripted variables and READ-DATA to enter information into a table.

Exercise: Using READ and DATA to fill a string vector. A vector is information arranged as an ordered collection of values.

```
T:  NEW
T:  99 DIM MN$(12)
T:  100 FOR I=1 TO 12
T:  110 READ MN$(I)
T:  120 DATA"JAN","FEB","MAR","APRIL","MAY","JUNE"
T:  125 DATA "JULY","AUG","SEPT","OCT","NOV","DEC"
T:  130 PRINT MN$(I)
T:  140 NEXT I
T:  RUN
R:  JAN
R:  FEB
    .
    .
    .
R:  DEC
```

This program sets up the array MN$ with the names of the months and prints each one out. To verify that the fifth month is MAY, try:

```
T:  PRINT MN$(5)
R:  MAY
```

Exercise: Use READ and DATA to fill a numeric array (a collection of numbers):

```
T:  99 DIM ND(12)
T:  110 READ ND(I)
T:  120 DATA 31,28,31,30,31
T:  125 DATA 30,31,31,30,31,30,31
T:  130 PRINT ND(I)
T:  RUN
R:  31
R:  28
    .
    .
    .
R:  31
```

This program puts the number of days in each month into the array ND and then prints out the number of days. To verify that the fifth month (MAY) has 31 days, try:

```
T:  PRINT ND(5)
R:  31
```

XV. PROGRAM DESIGN

The program development cycle was described in Chapter I. An important step in this cycle is to design the solution. In Chapters II through XIV, various features of VIC BASIC were demonstrated. No designs were offered because the exercises were more experiments than programs. In the next two chapters on VIC Color and Sound, programs will be written using some of the programming tools you learned in earlier chapters. Instead of presenting finished programs for you to figure out, the design process will be described.

There are two parts to any program design:

1. the procedure or steps required,
2. the data to be processed.

These two pieces need to be determined in parallel. You may not need to write sound or color programs for your computer, but go through the program design and programming examples anyway. You will find these techniques useful regardless of the problem. The program design process can be broken down into seven steps.

1. Define the problem.
2. Set down the high level (general) step-by-step solution to this problem.
3. Specify the information needed by the procedures (processes) described in step 2.
4. Decide in general what should be done by the program and what should be done by the person using the program.
5. Refine the current solution a relatively small amount.
6. Refine the definition of the information to match (support) the procedural solution found in step 5.
7. If the solution and the information (data) have been refined so that the code and data structures are obvious, start on the coding phase. If not, repeat from step 4.

A. Psuedo Code

To do a design, tools are needed. The tool that will be used is psuedo code, sometimes described as structured English. Psuedo code is more readable and understandable than either BASIC or flowcharts. It is used to give visual clues to the structure of the process. The English portion of psuedo code describes

WHAT needs to be done and not HOW it is to be done. Instead of using symbols and connecting lines (like flowcharts), psuedo code uses keywords and indentation to display the organization of a process. Psuedo code is easier to read because there are no branches (no GO TOs). Therefore, it can be read from top to bottom.

Psuedo code is not a programming language because it will not run on any computer. It is used to decribe the transition from a vague statement of the problem to the BASIC program solution. This gradual transition, helps to identify what really is required and simplifies the program writing. It is easy to transcribe psuedo code into BASIC or other programming languages. The difficult task is to decide what the program is supposed to do. Psuedo code helps you decide what is needed.

Quickly skim the definitions of the different legal psuedo code statements. Then go on to Chapter XVI where these statements are used to develop a color selection program. The example should help you understand the definitions.

1. General Form. The general form of a psuedo code description of a problem solution consists of keywords and text. The keywords are identified by all capital letters. The text is written in combined upper and lower-case. The keywords used are: SELECT, ENDSELECT, OTHERWISE, REPEAT,and ENDREPEAT.

2. Sequence. A sequence of actions is described using only text, no keywords are needed. For example, the statement 'Display data in error' is a legitimate psuedo code description of an action or sequence of actions needed to solve a part of a problem.

3. Conditional. A conditional psuedo code statement allows you to select among different actions. The keywords SELECT, OTHERWISE, and ENDSELECT are used. The general form is:

```
SELECT on condition (specify)
(condition is case 1)
   psuedo code statement(s)
(condition is case 2)
   psuedo code statement(s)
(condition is case n)
   psuedo code statement(s)
(OTHERWISE)
   psuedo code statement(s)
ENDSELECT
```

The general form is interpreted as: if the condition matches case 1, do the thing(s) specified in the scope of case 1. The scope of case 1 is all the psuedo code statements between (condition is case 1) and (condition is case 2). Likewise, if the condition matches case i, do the thing(s) specified in the scope of case i. If none of the conditions match, do the things specified in the scope of OTHERWISE. That is, all the psuedo code statements between OTHERWISE and ENDSELECT. Regardless of which scope is processed, the next thing to do is found after the ENDSELECT.

There are two simpler cases:

```
SELECT on condition (specify)
(condition is true)
   psuedo code statement(s)
ENDSELECT
```

This simplified case says, if the condition is true do something. If the condition is not true (false), don't do anything. The other simple case is:

```
SELECT on condition
(condition is true)
   psuedo code statement(s)
(OTHERWISE)
   psuedo code statement(s)
ENDSELECT
```

This form says, if the condition is true do one set of things and if the condition is false, do something else. In either case, the next thing to be done is found after the ENDSELECT.

4. Repetition. A repetition psuedo code statement allows you to specify that an action needs to be done a variable number of times. The steps (actions) may be done many times, a few times, once, or not done at all. It all depends on the condition. REPEAT and ENDREPEAT are the keywords. The form of repetition is:

```
REPEAT until condition
   psuedo code statement(s)
ENDREPEAT
```

Everything that is within the scope of the REPEAT (psuedo code statements between the REPEAT and the matching ENDREPEAT) are done over and over until the condition is met (is true).

5. Indentation. You will notice in the description of selection and repetition all statements within a single scope were indented the same amount. Indentation shows that the statements indented are related to the keyword with indentation one level less. For example, let's describe a simple estimating situation. There are several jobs to be estimated. Some can be done from a standard book, others require the boss to do the estimating.

```
REPEAT until all job estimates are printed
   Prepare job estimate cover sheet
   SELECT on type of job
   (job type is standard)
      Get book
      Prepare estimate from book
      Print estimate
   (job type is special)
      Ask boss to prepare estimate
      Print estimate
   ENDSELECT
ENDREPEAT
```

The first level of indentation is used by REPEAT and ENDREPEAT. Everything indented more than the first level is within the scope of the REPEAT. The keywords SELECT and ENDSELECT are at the second level of indentation. All statements indented more that level 2 are within the scope of the SELECT. The line (job type is standard) is also at level 2. It marks the beginning of one of the two (in this case) choices. The statements following (job type is standard) that are indented more than level 2 are part of that choice. The next line at level 2, (job type is special), marks the end of the first choice.

This is a fairly simple example, but you can see that even it is less understandable, if written without indentation.

```
REPEAT until all job estimates are printed
Prepare job estimate cover sheet
SELECT on type of job
(job type is standard)
Get book
Prepare estimate from book
Print estimate
(job type is special)
Ask boss to prepare estimate
Print estimate
ENDSELECT
ENDREPEAT
```

You should use some form of indentation to help describe the structure of the psuedo code. The number of spaces to indent is arbitrary. Three spaces is a good compromise. It is hard to notice the difference in indentation levels if less than three spaces are used. More than three spaces wastes paper and makes it difficult to fit deeply nested psuedo code on a 65-75 character line.

B. Data Description

The description of the data parallels the description of the procedures (psuedo code). The data description includes sequence, selection, and repetition. The data is described using text and a few special characters. No key words are needed to describe data. This description is sometimes called a data dictionary.

1. Sequence. A sequence of data items is described with text and the + (plus) sign. For example:

```
Color_mix=Background_color+Border_color
          +Character_color
```

is read as Color mix is Background color AND Border color AND Character color. All three items are needed to have Color mix.

2. Selection. Selection of data means a choice between different possible data items. The special symbols [, |, and] are used to show that a choice exists. The [marks the beginning of the choices. The | separates individual choices and the] marks the end of the choices. The general form is:

```
This_data=[ One type of data |
            Another type     |
            A third  ]
```

This means that one of the three data items is to be selected. It can be read as This data is One type of data OR Another type OR A third.

Just as there are two special cases of selection of procedure steps, there are two special cases of data selection. The first simple case is the choice between exactly two alternatives.

```
Choice = [ One | Other ]
```

This is read as Choice is One OR Other. The '|' is used to separate the two data items. The second special case is:

```
Option = [ Only one ]
```

This form of the selection data dictionary entry means Option is either Only one OR nothing at all. This form is used for optional data items.

3. Repetition. A data dictionary entry that specifies repetition allows you to show that a data item needs to be included a variable number of times. The data may appear many times, a few times, once, or not at all. The number of times depends on the limits shown and on the actual data. The general form is:

```
Name = Ø < Title > 5 + First_Name + Last_name
```

This is read as Name is Ø to 5 occurrences of Title AND First name AND Last name. The first number is the lower limit, the < signals the start of the repetition, the > marks the end of the repetition, and the last number is the upper limit. You can think of the < and > as specifying the scope of the repetition.

Example: Describe a VIC character string.

```
VIC_character_string=Ø<character>255
character = 8 < binary digit > 8
binary_digit = [ Ø | 1 ]
```

This set of data dictionary entries says that a VIC character string is made up of from Ø to 255 characters, each character is exactly eight (8) binary digits, and each binary digit is either a Ø or a 1.

C. Typical BASIC Implementation

The psuedo code statements must be translated to any programming language available on your machine. On the VIC that can be either BASIC or machine language. Machine language will not be covered in this book. Psuedo code can be translated to BASIC in several different ways.

1. Sequence. A sequence of statements can be any simple statement, such as, replacement, input, or output.

Example: Sequence statements

```
A=5
READ B
PRINT#1,C$
```

2. Selection. Control statements, such as, IF, ON X GOTO, or ON X GOSUB are used to implement selection.

Example: Choice of four alternatives.

```
ON I GOTO 100,200,300,400
```

Special case of choice of two:

```
100 IF A=0 THEN B=5:GOTO 200
110 B=6
200 ...
```

Special case of option:

```
300 IF B=5 THEN C=A
```

3. Repetition. Repetition can be done in several different ways. If the repetition is regular, i.e. repeats in a patern some number of times, you can use a FOR - NEXT loop.

Example: Regular repeat loop

```
100 FOR I=1 TO 50
110 A(I)=I
120 NEXT I
```

If the repetition is conditional, as in a table search, it can be done with an IF statement and a GOTO.

Example: Conditional repetition

```
100 I=0
110 I=I+1
120 IF I>LIMIT THEN 200
130 IF A(I)=TEST THEN 300
140 GOTO 110
200 REM NOT FOUND
300 REM FOUND-Ith ELEMENT MATCHES
```

If the repetition is until an operator signals the end, it can be done in two ways on the VIC.

Example: Operator types 'STOP' 'RESTORE' to end

```
100 INPUT D
  ...
  ...
  ...
200 GOTO 100
```

Example: Operator types 'end' to terminate.

```
100 INPUT D$
110 IF D$="END" THEN STOP
120 D=VAL(D$)
  ...
  ...
  ...
200 GOTO 100
```

In the following chapters on color and sound, more examples are given to show you how to translate from psuedo code to BASIC.

XVI. COLOR

In this chapter you will use the techniques of program design described in Chapter XV to develop a program. This program will demonstrate the color combinations available on your VIC.

A. Statement of the Problem

The problem to solve is to design a program to set up the VIC screen with the user's choice of colors. On the top (most general) level the description of a psuedo code solution is:

```
Setup needed data
REPEAT until operator finds desired combination
   Get choice of colors
   Set VIC to give those colors
ENDREPEAT
```

This procedure needs input to specify the colors. It also needs information on how to set the VIC to a particular color.

Several questions need to be answered before the next step (refinement of solution) can be done. These questions will help clarify the design and determine what should be done by the program and what should be done by the user.

1. What choices of colors are you going to provide? Let's assume that background, border, and character color are all selectable.

2. How are errors to be handled? Is it permissible to ask for a black character on a black background? Let's assume that a nonexistant color choice or an invisible one (red on red) gives an error message. After the error message you will display a list of acceptable colors for this type. Type is background, border, or character.

3. How should colors be specified? The VIC has keys labeled with color names. That is one good way to select colors. Typing B L A C K or G R E E N is a natural way to specify color. People who don't like to type might prefer abreviations like BLK or GRN. Let's provide all three ways to enter choice of color.

B. Refinement of the Solution

Based on the top level design and the answers to questions 1-3, you can start expanding (refining) the design. As more (detailed) questions come up, you will use them to further refine the design. Let's start on the second level description by refining the first level statement 'Get choice of colors.' You can use the answer chosen for question #1 to help specify level two.

```
Get background color
Get border color
Get character color
```

These three lines are the equivalent of the original and they are a very small step closer to a complete description of the problem to solve.

The information required can be clarified now. The program must be able to distinguish between background, border, and character color input. It also must be able to set the VIC's background, border, and character colors.

So far, you have only used the sequence of psuedo code statements introduced in Chapter XV. No keywords have been needed yet. As you proceed to the next (third) level, repetition and selection will be needed to further refine this part of the solution. On the third level, let's use what we have and the answer to question two to expand the 'gets'. Then 'Get background color' can be described by:

```
REPEAT until legal background color entered
   Input background color
   SELECT on legal background color
   (illegal background color)
      Display error message
      Display legal background colors
   (legal background color)
      Save this legal background color
   ENDSELECT
ENDREPEAT
```

This sequence of psuedo code describes in more detail what is needed to 'Get background color' and it shows what should be done in case of an error.

The information needed to support this step is a list of legal background colors. It is used for the test of legal colors and for the error situation to display what the user should have typed.

On the fourth level, using question 3, you can refine (be more precise about) 'Input background color'.

```
Prompt operator for background color
Get color info from keyboard
```

Since the VIC works with numbers and will need numbers to set the color, let's convert full names, and abreviations to their corresponding number. You can do part of your testing for legal data in the table search and part based on the number. Expanding 'Get color info from keyboard' with this in mind gives

```
Read character string from keyboard
SELECT on content of string
(String is number)
   Set color number to string value
   SELECT on number
   (number is not legal)
      Set color number to standard illegal value
   ENDSELECT
(string is abreviation for color)
   Set color number to equivalent abreviation
(string is full name of color)
   Set color number to equivalent full name
(OTHERWISE)
   Set color number to illegal
ENDSELECT
```

Note that the OTHERWISE only catches some of the illegal colors. The rest will be caught later in the invisible character test section.

The three (3) pieces of information needed for these steps are: prompts for the operator, a table of abreviations, and a table of full color names.

The conditional SELECT based on illegal background color can be done in several ways. A simple way is to check to see if the color name is in the list of legal background color names. You can refine the 'Set color number to equivalent full name' by replacing it with a REPEAT loop that searches through the table of legal background colors. You also need a search of the abreviation table.

Question #4- Do you want to restrict all the color choices to the eight (8) basic colors? Or, do you want to allow all possible (background) colors? Let's make this routine general and allow all possible colors to be specified. The extended background colors will not have single labeled keys assigned, but can be entered as a number, a full name (LIGHT BLUE), or an abreviation (LTB). These extended colors will be illegal if specified for border or character color. The data ,therefore, must contain information on which colors are legal for each type. The following table search could be used for the full name table and a similar search could be done on the abreviation table.

```
Set color is not found
Start at first entry in background color table
REPEAT until all colors checked OR
       color is found
   SELECT on entered color
   (Yes- color is in background table)
      Set color is found
   ENDSELECT
   Move to next entry in color table
ENDREPEAT
```

The psuedo code for getting border color and character color is the same except for the name of the color. Since the psuedo code is so similar you can simplify the description by using repetition. Let's go back to level two where we had

```
Get background color
Get border color
Get character color
```

and replace that psuedo code with psuedo code that does the same thing but uses REPEAT instead of duplicating statements later. Notice that we are going to discard some of the work previously done and start over on a slightly different approach. At this point not much has been invested, so it is easy to start again. You must learn to start again when a better way is discovered. Don't be afraid to throw away early versions. Getting committed to one approach too early is a common mistake. Try to avoid it.

```
Start at first color type
REPEAT until all color types entered
   Get this color type choice
   Move to next color type
ENDREPEAT
```

This form of the solution is simplier and much more compact than the earlier approach.

C. Psuedo Code of Get Choice

It looks like all the pieces are now ready to give a full expansion of 'Get choice of colors'. You can use a generalized version of 'Get background color' to expand 'Get this color type choice'. Let's take all the levels from 'Get background color' and build a complete 'Get choice of colors'.

```
Start at first color type
REPEAT until all color types entered
   REPEAT until legal color for this type entered
      Input color for this type
      Set color is not found
      SELECT on keyboard input
      (keyboard input is number)
         SELECT on range of number
         (number is not in range)
            Set color number to standard illegal value
         ENDSELECT
      (keyboard input is full color name)
         Start at first entry in full color name table
         REPEAT until all colors checked OR
            color is found
            SELECT on entered color
            (Yes- color is in full name table)
               Set color number based on full name
               Set color is found
            ENDSELECT
            Move to next entry in full name table
         ENDREPEAT
      (keyboard input is abreviation)
         Start at first entry in abreviation table
         REPEAT until all colors checked OR
            color is found
            SELECT on entered color
            (Yes- color is in abreviation table)
               Set color number based on abreviation
               Set color is found
            ENDSELECT
            Move to next entry in abreviation table
         ENDREPEAT
      ENDSELECT
      SELECT on legal color for this type
      (illegal color for this type)
         Display error message
         Display legal colors for this type
      (legal color for this type)
         Save this legal color
```

```
      ENDSELECT
   ENDREPEAT
   Move to next color type
ENDREPEAT
```

D. Psuedo Code of Set VIC

The pseudo code for set VIC is quite a bit simplier, since most of the work was done to get the colors right. Now let's expand 'Set VIC to give those colors'. You still needed to take care of the invisible character case described in question number 2. The second level of 'Set VIC to give those colors' is:

```
SELECT on color combination
(color combination is legal)
   Convert colors to VIC form
   Set VIC to colors selected
(OTHERWISE)
   Display illegal color combination message
ENDSELECT
```

The data required to combine a legal background color with the border color will add two columns to our color table. To set the character color you must print something. Let's print 'DEMO' and the color choices.

E. Miscellaneous Psuedo Code

This section contains the psuedo code for the data initialization, error message display, and display settings.

1. Initialize. This sequence of psuedo code describes what must be available for the rest of the solution to work.

```
Save space for color codes (3), full color names,
     and abreviated color names
REPEAT until all colors entered
   Get full name
   Get abreviated name
   Start at first color value
   REPEAT until all color values entered
      Get color value
   ENDREPEAT
ENDREPEAT
Setup messages for background, border,
     and character messages
```

2. Display Error. Display error shows the incorrect data and the legal color names that can be entered for this color type.

```
Display bad color name
Display message text
Start at first entry in color table
REPEAT until all color names checked
   SELECT on color type
   (color is legal for this type)
      Display legal color abreviation
   ENDSELECT
   Move to next entry in color table
ENDREPEAT
```

3. Display settings. Here you will display on the screen the colors you set your VIC to. This message will give you a way to check the program and also check the tuning on your TV set. Depending on the particular TV you use you may find the colors are true.

```
Display 'DEMO'
Start with first color type
REPEAT until all color types displayed
   Display name of color type and color choice
ENDREPEAT
```

F. Data Description

The refinement of the data needed should parallel the refinement of the procedures. At the top level you have:

```
color_choice=background_color+border_color
             +character_color
```

Let's start breaking down each of the three color types. The first one to refine is background.

```
background_color=[legal_background_number |
                 legal_background_name |
                 legal_background_abreviation  ]
```

Now you need to determine what are a legal background numbers, names, and abreviations. You know from the development of the psuedo code that the border and character colors are very similar to the background color. There we made use of the similarity by reusing part of the psuedo code by setting up a REPEAT loop instead of repeating code. You can do the same sort of simplifying here. The common parts are the color

number, color name, and color abreviation. The differences can be identified by a flag that specifies whether this color is legal or not for this color type. Let's use this information to define the next level of data refinement.

```
legal_background_number=color_number+background_ok
legal_background_name=color_name+background_ok
legal_background_abreviation=color_abreviation
                              +background_ok
```

The border and character colors are done the same way.

```
legal_border_number=color_number+border_ok
legal_border_name=color_name+border_ok
legal_border_abreviation=color_abreviation
                          +border_ok
legal_character_number=color_number+character_ok
legal_character_name=color_name+character_ok
legal_character_abreviation=color_abreviation
                             +character_ok
```

The legal color numbers can be found on your VIC keyboard. They are the numbers 1 through 8. Writing that in our data dictionary format gives:

```
color_number=[1|2|3|4|5|6|7|8]
```

The permitted color names are listed in the guide provided with your VIC. In the first edition, the color names were listed on page 37.

```
color_name=[black|white|red|cyan|purple|
            green|blue|yellow|orange|lt orange|
            pink|lt cyan|lt purple|lt green|
            lt blue|lt yellow]
```

The abreviations for colors can be taken from the keyboard for the first eight (8) colors. After those are entered, you will have to makeup your own. If you don't like the ones listed here, change the ones you want. The only restriction is that they must be exactly three (3) characters long. If you want to change the length of color abreviations, you must change the program. The color abreviations are:

```
color_abreviation=[blk|wht|red|cyn|pur|grn|blu|
                   yel|orn|lor|pnk|lcy|lpu|lgr|
                   lbl|lye]
```

If you look closely at the order of the color numbers, color names, and color abreviations, you will notice

that the first all match (black). They are all in the same order. To keep them in the same order, the abreviation red was used even though it is no shorter than the full name.

The next step is to specify the color values. These are needed for the VIC to actually set the colors. In your VIC guide there is a section on the POKEs used to change colors. This color demo program will have to use the same POKEs. From the table of screen and border color combinations you can get the screen (background) color values. They are:

```
background_color_value=[0|16|32|48|64|80|96|112|
                        128|144|160|176|192|
                        208|224|240 ]
```

These background colors are listed in the same order as the color numbers, color names, and color abreviations.

The border color values can also be extracted from the table of screen and border combinations. It is shorter (only 8 long) because only the basic 8 colors are permitted for use in the border. Since we have 16 entries for color name and abreviation, let's make this 16 long to match. The last 8 will have an illegal color indicator. Then

```
border_color_value=[8|9|10|11|12|13|14|15|
                    8<[illegal flag]>8]
```

The values for character color can be found in Appendix J of the VIC guide. They are spread out through the character set. Let's put them into the same order as the color names and abreviations.

```
character_color_value=[144|5|28|159|156|30|31|158|
                       8<[illegal flag]>8]
```

Character colors are also restricted to the 8 basic colors. Just as you did with border colors, character color values are filled out with eight (8) illegal color value flags.

Now we have all the data that we need to build a complete color table. The color table will have 16 entries for legal color names and abreviations. The 17th entry in the table will be used as the standard illegal color. Writing that in data dictionary form:

```
color_table=16<color_entry>16+illegal_entry
```

Each color entry will be composed of all the information pertaining to that color.

```
color_entry=color_name+color_abreviation
            +background_value+border_value
            +character_value
```

The color type is the last data to be described.

```
color_type=[background|border|character]
```

G. Writing the Program

This approach to writing programs is based on the fact that it is easy to write short simple programs. What you have been doing is breaking a medium size program up into several short simple programs. This is the divide and conquer approach.

Before starting to write the program, let's assign preliminary statement numbers for the different sections of the program.

```
0100-0199   Initialization
0200-0300   Top level REPEAT loop
3000-3999   Get color choice subroutine
4000-4999   Set VIC to colors chosen
5000-       Data for initialization
```

The first program statements that we will write are for the main control loop. The 'REPEAT until operator finds the desired combination' statement can be done very simply. Just make an unconditional loop. You can always exit from a program (quit) by typing 'STOP' and 'RESTORE'. However, if you do make this loop unconditional and use 'STOP'/'RESTORE' to end it, the screen colors will be set back to the standard ones. This is a feature of the 'RESTORE' key. So, you need to put in a test for the end. Let's also put in a remark so that you know which program this is. A complete listing of this program is included in Appendix A.

```
T:  50 REM VIC COLOR DEMO BY TIS
T:  200 INPUT"CONTINUE (Y,N)";A$
T:  205 PRINT" 'CLR' "
T:  210 IFLEFT$(A$,1)="N"THENSTOP
T:  220 IFLEFT$(A$,1)<>"Y"THEN200
T:  250 GOTO200
```

For testing you can insert 230 PRINT"YES-OK". Test this portion with yes, no, and something else. Then add the initialization portion. Let's define all the data you need. Put it into the program and test that it is in properly. The psuedo code for initialization is:

```
Save space for color codes (3), full color names,
     and abreviated color names
REPEAT until all colors entered
   Get full name
   Get abreviated name
   Start at first color value
   REPEAT until all color values entered
      Get color value
   ENDREPEAT
ENDREPEAT
Setup messages for background, border,
     and character messages
```

The program statements needed to accomplish initialization are:

```
T:  100 NC=17:DIMCO%(NC,3),FU$(NC),AB$(NC)
T:  110 FORI=1TONC:READFU$(I),AB$(I)
T:  120 FORJ=1TO3:READCO%(I,J)
T:  130 NEXTJ
T:  150 NEXTI
T:  161 CC$(1)="BACKGROUND"
T:  162 CC$(2)="BORDER"
T:  163 CC$(3)="CHARACTER"
```

Before you can test this portion, you must enter the data for the read statements.

```
T:  5001 DATA BLACK,BLK,0,8,144
T:  5002 DATA WHITE,WHT,16,9,5
T:  5003 DATA RED,RED,32,10,28
T:  5004 DATA CYAN,CYN,48,11,159
T:  5005 DATA PURPLE,PUR,64,12,156
T:  5006 DATA GREEN,GRN,80,13,30
T:  5007 DATA BLUE,BLU,96,14,31
T:  5008 DATA YELLOW,YEL,112,15,158
T:  5009 DATA ORANGE,ORN,128,-1,-1
T:  5010 DATA LT ORANGE,LOR,144,-1,-1
T:  5011 DATA PINK,PNK,160,-1,-1
T:  5012 DATA LT CYAN,LCY,176,-1,-1
T:  5013 DATA LT PURPLE,LPU,192,-1,-1
T:  5014 DATA LT GREEN,LGR,208,-1,-1
T:  5015 DATA LT BLUE,LBL,224,-1,-1
T:  5016 DATA LT YELLOW,LYE,240,-1,-1
T:  5017 DATA ILLEGAL,ILL,-1,-1,-1
```

There are several interesting techniques in this data definition. Notice that the color number is the statement number minus 5000. This information will be useful while checking the program. The flag -1 is used to indicate that a particular color is illegal for border or character. The last data line (5017) contains an entry for illegal data and all three data values are set to -1. Let's test this portion of program by adding the following:

```
T:  140 PRINT I;FU$(I);AB$(I);CO%(I,1);CO%(I,2);CO%(I,3)
T:  145 GETA$:IFA$=""THEN 145
```

This temporary section of program will let you check the data. It will print each entry and wait for a key stroke. You can check each entry to make sure it was typed correctly. RUN the program to test it.

Assume that during your testing the last two lines printed are:

```
R:  6 GREENGRN  80  13  30
R:  ?TYPE MISMATCH
R:   ERROR IN 110
```

This tells you immediately to look at line 5007 because that is the next item to read. If you find that some of the data is wrong, you can find the statement number by adding 5000 to the first number (color number) on the display line. After checking to make sure all the data statements are correct, remove lines 140 and 145. They will not be needed any longer.

The next step is to enter the program statements for numeric input of colors. First, let's put the subroutine call into the main control loop and then add the subroutine.

```
T:  230 GOSUB3000
T:  3000 FORCC=1TO3
T:  3100 PRINT"ENTER ";CC$(CC);" COLOR"
T:  3110 INPUTA$
T:  3120 IFLEN(A$)<3THENCN=VAL(A$):GOTO3200
T:  3200 IFCN<1ORCN>NC-1THENCN=NC:GOTO3300
T:  3210 IFCO%(CN,CC)=-1THENCN=NC
T:  3300 IFCN<>NCTHEN3400
T:  3310 PRINTA$;" IS ILLEGAL ";CC$(CC);" COLOR"
T:  3350 GOTO3100
T:  3400 CV(CC)=CN
T:  3410 NEXTCC
T:  3420 RETURN
```

Test this with both legal and illegal color numbers. You will find that the numbers 9-16 are legal for the extended background colors. Even though the keyboard is not marked for these colors, this feature can be useful. Notice that the error message is not complete. Add the following lines to display the list of legal colors on error.

```
T:  3320 FORI=1TONC
T:  3325 IFCO%(I,CC)<>-1THENPRINTAB$(I);",";
T:  3330 NEXTI
T:  3340 PRINT"'CRSR-L' ARE OK"
```

The cursor left in line 3340 erases the last comma. Test illegal color numbers again. You should get the list of valid colors for a color type with the error message. To make sure that the right colors were chosen, add the following program statements to display what was typed in.

```
T:  240 GOSUB4000
T:  4000 IFCV(1)<>CV(3)THEN4030
T:  4010 PRINT"BACKGROUND AND CHARACTER CAN'T BOTH BE";FU$(CV(1))
T:  4020 RETURN
T:  4030 REM
T:  4060 PRINTTAB(9);"DEMO"
T:  4070 FORI=1TO3
T:  4080 PRINTCC$(I);" ";FU$(CV(I))
T:  4090 NEXTI
T:  4099 RETURN
```

Now test numeric input again. Do the color numbers typed match the color names displayed? If not, check the data statements at color number plus 5000. Be sure to try a test with the background color and the character color the same. The next part to add is for abreviated color names.

```
T:  3140 GOSUB3600
T:  3600 FORCN=1TONC
T:  3610 IFA$=AB$(CN)THENRETURN
T:  3620 NEXTCN
T:  3630 CN=NC
T:  3640 RETURN
```

Test this with some legal and illegal abreviations. The final step in the get color choice portion of the program is to add the statements for full color names.

```
T:  3130 IFLEN(A$)>3THENGOSUB3500:GOTO3200
T:  3500 FORCN=1TONC
T:  3510 IFA$=FU$(CN)THENRETURN
T:  3520 NEXTCN
T:  3530 CN=NC
T:  3540 RETURN
```

Use legal and illegal full names to test this portion. It would be a good idea to go back and test numeric and abreviated input to make sure that the new statements did not destroy parts that worked before. (They shouldn't have.) Be sure to try something like LT BLUE for character or border color. You should get an error response.

The final step is to add the program statements that actually set the VIC colors. It is possible to POKE addresses and values that would hang up the machine. If you make such a mistake, just press 'STOP' and 'RESTORE' to recover. If you are concerned about losing the program you spent so much time typing in, save a copy on cassette. After entering the next three lines, would be a good time to do so.

```
T:  4030 POKE36879,CO%(CV(1),1)+CO%(CV(2),2)
T:  4040 C$=CHR$(CO%(CV(3),3))
T:  4050 PRINTC$
```

That completes the color demo program. Test it with various combinations of numeric, abreviated, and full color names. What color combinations do you like best?

XVII. SOUND

The VIC has a sound generator that can make music and noises through the speaker on your television set. This chapter will use the program design techniques introduced in Chapter XV to develop a simple music composition program. These techniques were used to develop the color demonstration program in Chapter XVI. If you haven't already read Chapters XV and XVI, do so now. You will find this chapter easier to follow, if you read Chapters XV and XVI first.

A. The Problem

Let's develop a program to use the VIC sound generator feature. The program must let you enter a song, play a song, save a song on cassette, and load a song from cassette. The input needed to specify a song is the octave, the note, and the length of the note. When a song is saved on cassette, a name is needed to identify the song. This name will be used later when you want to retrieve the song from tape. Playing a song consists of starting at the first note and proceeding through each note until the end of song is reached. The notes will be specified in a natural way (c,a#,etc.) and it is up to the program to generate the correct data for the VIC.

B. Restrictions

The music composition program is intended to demonstrate the sound generation capabilities of the VIC. Some simplifications are necessary to get a working demonstration program running quickly. This program is not intended to be a professional song writers answer to a computerized composer.

The program will have the following restrictions:

1. Only single notes, not chords, will be allowed.
2. Each song will be played from beginning to end. There will be no way to play part of a song.
3. If the song has a chorus, it will have to be entered after each verse.
4. Once a song has been entered, the only way to change it is to reenter the entire song. There will be no way to correct or change an existing song.

You may want to expand this program later to eliminate some of these restrictions.

C. Sound Program Design

You can use the requirements described in section A, the restrictions listed in section B, and the tools from Chapter XV to layout the top level design of the sound program.

```
Initialize music data
REPEAT until no more music desired
   Get action request
   SELECT on action request
   (action is read)
      Read song from tape
   (action is write)
      Write song to tape
   (action is compose)
      Compose song
   (action is play)
      Play song
   (OTHERWISE)
      Display action error
   ENDSELECT
ENDREPEAT
```

This defines the problem in our psuedo code language. The next step is to start expanding the general statements at level one to describe in more detail what is desired.

D. Refinement of the Solution

Let's start refining the solution with the four (4) generalized statements of the actions. Then 'Read song from tape' can be described as:

```
Get name of song
Open tape file to read song
Start at first note of song
REPEAT until all notes read
   Get note value and duration from tape
   Save note value and duration in song
   Move to next note
ENDREPEAT
Close tape file
```

The expansion for 'Write song to tape' is almost exactly the same except for the direction of the data. It looks like this:

```
Get name of song
Open tape file to write song
Start at first note of song
REPEAT until all notes written
   Write note value and duration
   Move to next note
ENDREPEAT
Write end of song indicator
Close tape file
```

The 'Compose song' expansion is also similar. It looks like:

```
Start at first note
REPEAT until all notes entered
   Get octave, note, and duration from keyboard
   Convert octave and note to VIC note value
   Save note value and duration in song
   Move to next note
ENDREPEAT
```

The last of our four (4) actions is 'Play song'. It can be expanded like this:

```
Start at first note in song
REPEAT until all notes are played
   Get note value and duration from song
   Play note for duration
ENDREPEAT
```

The data needed for these four actions are octave, note, duration, end of song indication, the song (consisting of notes), and the VIC representation for octave and note (note value).

Let's further expand the psuedo code statement in 'Compose song' that says 'Get octave, note, and duration from keyboard'. You will need to decide on prompts and format of the input. The expansion of 'Get octave' is:

```
REPEAT until legal octave entered
   Prompt for octave
   Get octave from keyboard
   Start with first octave name
   REPEAT until all octave names have been checked OR
                octave name found
```

```
            SELECT on octave name entered
            (matches octave name in table)
               Set octave name found
               Save octave
            ENDSELECT
            Move to next octave name entry
         ENDREPEAT
      ENDREPEAT
```

The expansion of 'Get note' can be done in a similar way. It is:

```
      REPEAT until legal note entered
         Prompt for note
         Get note from keyboard
         REPEAT until all note names checked OR
                      note name found
            SELECT on note name entered
            (matches note name in table)
               Set note name found
               Save note
            ENDSELECT
            Move to next note name entry
         ENDREPEAT
      ENDREPEAT
```

The duration is specified in the same way. The expansion for 'Get duration' is:

```
      REPEAT until legal note length (duration) entered
         Prompt for note length
         Get note length from keyboard
         REPEAT until all note lengths checked OR
                      note length found
            SELECT on note length
            (note length found in table)
               Set note length found
               Save note length
            ENDSELECT
            Move to next note length entry
         ENDREPEAT
      ENDREPEAT
```

If you recall the development of the color demo in Chapter XVI will notice the similarity. In the color demo, you replaced the repeated program steps by generalizing and creating a REPEAT loop. In this program we will not generalize. This will show that there are different ways to create a program. A good future exercise is to convert this program to use a REPEAT loop for the different types of note input. Try that later after you get this version of the program working correctly.

The last psuedo code statement that needs to be expanded is in the 'Play song' section. The psuedo code statement for 'Play note for duration' expanded becomes:

```
POKE note value into sound generator 2
POKE zeros into all other sound generators
Turn volume on
Save current time
REPEAT until duration of note has passed
ENDREPEAT
Turn volume off
```

E. Data Definition

The data that is needed is the actions you can do, octaves, notes, durations, the composed song, and the VIC representation for the note values. The first data dictionary entry is for the actions.

```
action=[read|write|compose|play]
```

Let's now define octaves. There are three possible using a single sound generator. Call them lower, middle, and upper. Using our data dictionary format and a single letter to specify the octave gives:

```
octave=[ l | m | u ]
```

The VIC guide list 12 notes that can be identified. They are:

```
note=[ c|c#|d|d#|e|f|f#|g|g#|a|a#|b ]
```

These are the names to use for the notes. You can add r, for rest, so that you have a way to produce silence. A nice extension to this program would be to add the 'flat' notes. Try that after you get the basic program working. The next data to define is the duration of a note. Let's just use the simple durations of 1/16, 1/8, 1/4, 1/2, and whole notes. If you name these durations with single characters representing the time, you can write:

```
duration=[s|8|q|h|w]
```

The song can be represented by a collection of note values and durations. Because of memory limitations you need to set an upper limit on the length of any one song. Let's also leave room for using the other sound generators to make chords. For this version of the

program, the other two sound generators will be turned off. The data dictionary definition of song is:

song=Ø<note_value+space+space+duration_value>ms

In the song data dictionary entry, ms stands for the number of notes in the longest song (Maximum Song). The last data dictionary entry to define is the VIC representation for each of the possible notes. Because octave and note have been defined, let's organize the table of note values as 3 rows (one for each octave) by 12 columns (one for each note). A general description of note value is:

note_value=3<12<POKE#>12>3

The POKE# can be found in the VIC guide. When you defined the notes you started with c. The 3 POKE#s for c are 135,195, and 225. These POKE#s are in the same order as the octaves were defined, namely lower, middle, and upper. Taking the values from the VIC guide in the same order for the rest of the notes gives:

```
POKE#=[ 135 | 195 | 225 |
        143 | 199 | 227 |
        147 | 201 | 228 |
        151 | 203 | 229 |
        159 | 207 | 231 |
        163 | 209 | 232 |
        167 | 212 | 233 |
        175 | 215 | 235 |
        179 | 217 | 236 |
        183 | 219 | 237 |
        187 | 221 | 238 |
        191 | 223 | 239  ]
```

That completes the major data items needed for the VIC sound generator demonstration program.

F. Writing the Program

With all of the psuedo code already written, you can block out the statement numbers for the various parts of the program. A complete listing of the sound demonstration program is included in Appendix A. After entering a part of this program, save it on tape. Do this after each major portion has been typed in. That way, if you accidently type NEW or have the power go off, you won't lose much typing. For statement numbers let's use:

```
0100-0199  Initialization
0200-0499  Top level REPEAT loop
1100-1199  Read song action
1200-1299  Write song action
1300-1399  Compose song action
1400-1499  Play song action
3000-3999  Input prompts
4000-4999  Keyboard input
5000-      Data for initialization
```

The first statements that you will write are for the main control loop. The 'REPEAT until no more music desired' can be done very simply. Just make an unconditional loop. You can always exit from a program by typing 'STOP' and 'RESTORE'. This program is different than the color demonstration because you do not want to keep the final state of the program. Let's also put in a remark so that you know which program this is.

```
T:  50 REM VIC SOUND DEMO BY TIS
T:  200 PRINT" 'CLR' "
T:  410 GOTO 200
```

The next program statements to enter are for initialization. You need to save space for the song, the octave names, the note names, the names of the durations (note lengths), and all of the note values. You also need to define the number of actions, each of the VIC sound generator addresses, and the volume.

```
T:  100 MS=40:NN=12:NO=3:DIMSO%(MS,4),OC$(NO),NO$(NN)
T:  102 NL=5:DIMNL$(NL),NL(NL),SC%(NN,NO)
T:  105 NA=4:FORI=1TO5:S(I)=36873+I:NEXTI:VO=12
```

Now you can setup the note names and the note values for each octave for each note. There are two parts to this. The first part is the read statements and the second part is the data that goes with the read statements. Let's enter both parts now.

```
T:  110 FORI=0TONN:READNO$(I)
T:  120 FORJ=1TONO:READSC%(I,J)
T:  130 NEXTJ
T:  150 NEXTI
```

```
T:  5000 DATAR,0,0,0
T:  5001 DATAC,135,195,225
T:  5002 DATAC#,143,199,227
T:  5003 DATAD,147,201,228
T:  5004 DATAD#,151,203,229
T:  5005 DATAE,159,207,231
T:  5006 DATAF,163,209,232
T:  5007 DATAF#,167,212,233
T:  5008 DATAG,175,215,235
T:  5009 DATAG#,179,217,236
T:  5010 DATAA,183,219,237
T:  5011 DATAA#,187,221,238
T:  5012 DATAB,191,223,239
```

To test this portion of the program and make sure that the data has been entered correctly, add these two lines:

```
T:  140 PRINTI;NO$(I);SC%(I,1);SC%(I,2);SC%(I,3)
T:  145 GETA$:IFA$=""THEN145
```

After you have tested the data for note name and note values, remove lines 140 and 145. Now let's enter the names of the three parts of a note (octave, name, and length), the four actions (read, write, compose, and play), and the three octaves (lower, middle, and upper).

```
T:  161 CC$(1)="OCTAVE"
T:  162 CC$(2)="NOTE"
T:  163 CC$(3)="LENGTH"
T:  171 AC$(1)="READ"
T:  172 AC$(2)="WRITE"
T:  173 AC$(3)="COMPOSE"
T:  174 AC$(4)="PLAY"
T:  181 OC$(1)="L"
T:  182 OC$(2)="M"
T:  183 OC$(3)="U"
```

The next initalization information to enter is the names of the different note lengths and their corresponding values. The numbers used are based on 2 seconds for a whole note and 1/30 of a second (NL(0)) between notes. If you want to change the note timing, change the following numbers.

```
T:  190 NL$(0)="":NL(0)=(1/30)*60
T:  191 NL$(1)="S":NL(1)=(1/16)*120
T:  192 NL$(2)="8":NL(2)=(1/8)*120
T:  193 NL$(3)="Q":NL(3)=(1/4)*120
T:  194 NL$(4)="H":NL(4)=(1/2)*120
T:  195 NL$(5)="W":NL(5)=1*120
```

That completes the initialization of the information needed by the sound generation demonstration program. The next step is to add the statements for 'Get action request' and the 'SELECT on action request'.

```
T:  300 PRINT"ENTER COMMAND"
T:  310 INPUTA$
T:  320 FORI=1TONA
T:  330 IFAC$(I)=A$THEN400
T:  340 NEXTI
T:  350 PRINT "LEGAL COMMANDS ARE:"
T:  360 FORI=1TONA:PRINTAC$(I):NEXTI
T:  370 GOTO 300
T:  400 ONIGOSUB1100,1200,1300,1400
```

This section of program searches the table of legal actions. If a matching action is not found, it displays the legal actions and asks for you to try again. If a matching action is found, the subroutine at 1n00 is called. The n is the number of the action. Using statement numbers that reflect the organization of the program makes it easier to debug and modify the program. Try to develop your own techniques for program organization. The next part of the program to enter is the 'Compose song' portion. The outer loop part of that is 'REPEAT until all notes entered'.

```
T:  1300 IP=0
T:  1350 GOSUB3000
T:  1360 IFOK=1THENRETURN
T:  1370 IFIP<MSTHEN1350
T:  1380 PRINT"SONG DONE. NO MORE ROOM":RETURN
```

This section repeatedly gets notes until there are no more or until you run out of room. The work of getting the notes is done by the subroutine at 3000. Let's enter it next. It starts with 'Get octave'.

```
T:  3000 CC=1:OK=0:PRINT" 'CLR' NOTE";IP+1
T:  3010 GOSUB4000:IFA$="END"THENOK=1:RETURN
T:  3020 IP=IP+1
T:  3040 FORCN=1TONO
T:  3050 IFA$=OC$(CN)THEN3100
T:  3060 NEXTCN
T:  3070 PRINT"ERROR LEGAL ";CC$(CC);" ARE:"
T:  3080 FORI=1TONO:PRINTOC$(I):NEXTI
T:  3090 IP=IP-1:GOTO3010
```

This section uses the subroutine at 4000 to prompt for and get the name of the octave as input. To test this portion of the program add the subroutine at 4000 and the 2 lines of program to complete the subroutine at 3000.

```
T:  4000 PRINT"ENTER ";CC$(CC)
T:  4010 INPUTA$:RETURN
T:  3100 REM
T:  3999 RETURN
```

Test this portion of the program. Remember that you can only compose at this time and you should get undefind statement number errors, if you try other actions. After testing 'Get octave', you can enter the statements for 'Get note'. They are:

```
T:  3100 CC=2:GOSUB4000
T:  3110 FORI=0TONN:IFA$=NO$(I)THEN3200
T:  3120 NEXTI
T:  3130 PRINT"BAD NOTE. LEGAL ARE:";
T:  3140 FORI=0TONN:PRINTNO$(I);" ";:NEXTI:PRINT
T:  3150 GOTO 3100
```

The song has four parts four each note. SO%(NOTE,1) and SO%(NOTE,3) are the spaces for expansion to chords. They are always zero in this version. SO%(NOTE,2) is the single note to play and SO%(NOTE,4) is the duration for that note. The psuedo code statements for 'Convert octave and note to VIC note value' and 'Save note value in song' can be done in one BASIC statement.

```
T:  3200 SO%(IP,2)=SC%(I,CN)
```

The variable I contains the number of the note and the variable CN contains the number of the octave, when line 3200 is executed. With that information, all you needed to do is look up the note value in the table that you initialized in line 120. Converting to the VIC note value is actually very easy (with the right choice of data representation).

Now you can continue your testing by entering both octave and note. Try both legal and illegal values for octave and note. The next program steps to enter are for 'Get duration' and 'Save duration in song'. They are:

```
T:  3210 CC=3:GOSUB4000
T:  3220 FORI=1TONL:IFA$=NL$(I)THEN3300
T:  3230 NEXTI
T:  3240 PRINT"BAD TIME. LEGAL ARE:";
T:  3250 FORI=1TONL:PRINTNL$(I);" ";:NEXTI:PRINT
T:  3260 GOTO 3210
T:  3300 SO%(IP,4)=NL(I)
```

To allow for future expansion you saved space in the song for two more notes. This will let you add chords at some future time. Let's make sure that only a single note is entered in this version of the sound demonstration program by setting the other notes to silence. You can do that with the single line:

```
T: 3310 SO%(IP,1)=0:SO%(IP,3)=0
```

That completes the 'Compose song' portion of the program. Next let's enter the program statements for 'Play song'.

```
T:  1400 IFIP=0THENRETURN
T:  1410 FORI=1TOIP:FORJ=1TO3:POKES(J),SO%(I,J)
T:  1420 NEXTJ
T:  1425 POKES(5),VO
```

This sets each of the three sound generators to the note values saved in song and turns the volume on. Now you must time how long this note plays. You do that with:

```
T:  1430 X=TI+SO%(I,4)
T:  1440 IFTI<XTHEN1440
T:  1450 POKES(5)=0
T:  1460 X=TI+NL(0)
T:  1470 IFTI<XTHEN1470
T:  1480 NEXTI
T:  1490 RETURN
```

Line 1430 calculates when the note will be finished. Line 1450 turns the volume off. A slight pause (1/30 of a second) is provided between notes by lines 1460 and 1470.

This completes the 'Play song' portion of the program. Test it by entering your favorite melody. Remember you still have to add the program statements to save the song on tape. This means that you probaly want to keep the test songs relatively short. When you have tested the compose and play actions thoroughly, you can add the 'Write song to tape' portion. It is:

```
T:  1200 INPUT"SONG NAME";A$
T:  1210 OPEN1,1,1,A$
T:  1215 IFIP=0THEN1230
T:  1220 FORI=1TOIP:FORJ=1TO4:PRINT#1,SO%(I,J):NEXTJ,I
T:  1230 PRINT#1,-1:CLOSE1
T:  1240 RETURN
```

Line 1230 writes the end of song signal (-1) on the cassette tape. When you test the write song to tape action, make sure that the tape that you use doesn't have any valuable programs or data on it. The next (and final) step is to enter the action subroutine for 'Read song from tape'.

```
T:  1100 INPUT"SONG NAME";A$
T:  1110 OPEN1,1,0,A$
T:  1120 IP=0
T:  1130 FORI=1TO4:INPUT#1,X
T:  1140 IFX<0THEN1180
T:  1150 SO%(IP+1,I)=X:NEXTI
T:  1160 IP=IP+1:IFIP<MSTHEN1130
T:  1170 PRINT"SONG TOO LONG"
T:  1180 CLOSE1:RETURN
```

Line 1160 is important because it insures that the song read fits into the space that you set aside for it. Its is good programming practice to check that the data is within range. That completes the sound demonstration program. After testing it, save your final copy on tape. You can then compose and play songs, if you are a musician. However, if you are a programmer, you will have a hundred ideas about how to improve this program. In either case, enjoy your VIC!

APPENDIX A

PROGRAM LISTINGS

You may notice that the spacing in these listings is different than in the text. In the text more blanks were used to make the program more readable. These listings have most unneeded spaces removed. They are slightly more difficult to read, but they take less room in memory and execute slightly faster. Both the programs in the text and those in the Appendix should run the same.

A. A complete listing of the final version of the mailing list skeleton program (MAIL4).

```
100 PRINT"MAILING LIST SKELETON"
101 AT$(1)="ADD"
102 AT$(2)="CHANGE"
103 AT$(3)="PRINT"
104 AT$(4)="COUNT"
110 INPUT"ENTER ACTION";A$
112 IF A$="" THEN STOP
115 FORA=1TO4
120 IFA$<>AT$(A)THEN135
130 ON A GOTO 1000,2000,3000,4000
135 NEXTA
140 PRINT"INPUT MUST BE LEGAL!"
150 GOTO110
1000 PRINT"ADD SELECTED"
1990 GOTO150
2000 PRINT"CHANGE SELECTED"
2990 GOTO150
3000 PRINT"PRINT LIST REQUESTED"
3990 GOTO150
4000 PRINT"COUNT REQUESTED"
4990 GOTO150
```

B. Program to convert decimal number to binary character string and display (DEC-BIN).

```
90 REM DISPLAY BINARY VALUE
100 INPUT"ENTER DECIMAL NO.";R
430 R$=""
440 GOSUB2000
500 PRINT"DECIMAL";R;" IS"
510 PRINT"BINARY ";R$
520 PRINT
530 GOTO100
```

```
2000 REM CONVERT NO. TO BINARY CHARACTER STRING
2010 C$(0)="0":C$(1)="1"
2020 R2=R
2030 T2=INT(R2/2)
2040 S=R2-T2*2
2050 R$=C$(S)+R$
2060 R2=T2
2065 IF LEN(R$)>16THENR$= RIGHT$ (R$, 16):
     RETURN
2070 IFT2<>0THEN2030
2080 RETURN
```

C. Program to convert binary character string to a decimal number and display it (BIN-DEC).

```
90 REM DISPLAY DECIMAL VALUE
100 PRINT"ENTER BINARY NO"
105 INPUT N$
106 IF N$="" THEN STOP
110 T1$=N$
120 GOSUB1100
130 IFT<0THEN100
140 PRINTN$;"BIN"
145 PRINT "IS";T; "DECIMAL"
150 GOTO100
430 R$=""
440 GOSUB2000
500 PRINT"DECIMAL";R;"IS"
510 PRINT "BINARY ";R$
520 PRINT
530 GOTO100
1100 REM CONVERT BIN. CHAR. STRING TO DEC. NO.
1105 T=0
1110 L1=LEN(T1$)
1120 FORI1=1TOL1
1125 T=T*2
1130 IFMID$(T1$,I1,1)="1"THENT=T+1:GOTO1150
1140 IFMID$(TI$,I1,1)<>"0"THENT=-1:RETURN
1150 NEXTI1
1160 RETURN
2000 REM CONVERT NO. TO BINARY CHARACTER STRING
2010 C$(0)="0":C$(1)="1"
2020 R2=R
2030 T2=INT(R2/2)
2040 S=R2-T2*2
2050 R$=C$(S)+R$
2060 R2=T2
2065 IF LEN(R$)>16THENR$=RIGHT$(R$,16):RETURN
2070 IFT2<>0THEN2030
2080 RETURN
```

D. Complete listing of the display logical function program (LOGIC).

```
90 REM DISPLAY LOGICAL FUNCTIONS
100 PRINT"ENTER BINARY NO"
105 INPUT N$
106 IF N$="" THEN STOP
110 T1$=N$
120 GOSUB1100
130 IFT<0THEN100
140 NE=T
150 PRINT"ENTER LOGICAL FUNCTION"
160 INPUT "(A,O,N)";OP$
190 F$=LEFT$(OP$,1)
200 IFF$<>"A"ANDF$<>"O"ANDF$<>"N"THEN150
210 TW$=""
220 IFF$="N"THEN420
300 PRINT"ENTER SECOND BINARY NO"
305 INPUT TW$
310 T1$=TW$
320 GOSUB1100
330 IFT<0THEN300
340 TWO=T
400 IFF$="A"THEN R=NE AND TWO
410 IFF$="O"THEN R=NE OR TWO
420 IFF$="N"THEN R=NOT NE
430 R$=""
440 GOSUB2000
500 IFF$="N"THEN PRINT OP$;"b";N$;"bIS"
505 IFF$<>"N" THEN PRINT N$;"b";OP$;"b";TW$;"bIS"
510 PRINT R$
520 PRINT
530 GOTO100
1100 REM CONVERT BIN. CHAR. STRING TO DEC. NO.
1105 T=0
1110 L1=LEN(T1$)
1120 FORI1=1TOL1
1125 T=T*2
1130 IFMID$(T1$,I1,1)="1"THENT=T+1:GOTO1150
1140 IFMID$(T1$,I1,1)<>"0"THENT=-1:RETURN
1150 NEXTI1
1160 RETURN
2000 REM CONVERT NO. TO BINARY CHARACTER STRING
2010 C$(0)="0":C$(1)="1"
2020 R2=R
2030 T2=INT(R2/2)
2040 S=R2-T2*2
2050 R$=C$(S)+R$
2060 R2=T2
2065 IF LEN(R$)>16 THEN R$=RIGHT$(R$,16):RETURN
2070 IFT2<>0THEN2030
2080 RETURN
```

E. Complete listing of the color demonstration program. Note that the listing was printed directly from the program. This was done to avoid typing errors.

```
50 rem vic color demo by tis
100 nc=17:dimco%(nc,3),fu$(nc),ab$(nc)
110 fori=1tonc:readfu$(i),ab$(i)
120 forj=1to3:readco%(i,j)
130 nextj
150 nexti
161 cc$(1)="background"
162 cc$(2)="border"
163 cc$(3)="character"
200 input"continue (y,n)";a$
205 print"■"
210 ifleft$(a$,1)="n"thenstop
220 ifleft$(a$,1)<>"y"then200
230 gosub3000
240 gosub4000
250 goto200
3000 forcc=1to3
3100 print"enter ";cc$(cc);" color"
3110 inputa$
3120 iflen(a$)<3thencn=val(a$):goto3200
3130 iflen(a$)>3thengosub3500:goto3200
3140 gosub3600
3200 ifcn<1 or cn>nc-1 thencn=nc:goto3300
3210 ifco%(cn,cc)=-1thencn=nc
3300 ifcn<>ncthen3400
3310 printa$;" is illegal ";cc$(cc);" color"
3320 fori=1tonc
3325 ifco%(i,cc)<>-1thenprintab$(i);",";
3330 nexti
3340 print"■■ are ok"
3350 goto3100
3400 cv(cc)=cn
3410 nextcc
3420 return
3500 forcn=1tonc
3510 ifa$=fu$(cn)thenreturn
3520 nextcn
3530 cn=nc
3540 return
3600 forcn=1tonc
3610 ifa$=ab$(cn)thenreturn
3620 nextcn
3630 return
4000 ifcv(1)<>cv(3)then4030
4010 print''background and character can't both be'';fu$(cv(1))
4020 return
4030 poke36879,co%(cv(1),1)+co%(cv(2),2)
4040 c$=chr$(co%(cv(3),3))
4050 printc$
4060 printtab(9);"demo"
4070 fori=1to3
4080 printcc$(i);" ";fu$(cv(i))
4090 nexti
```

```
4099 return
5001 data black,blk,0,8,144
5002 datawhite,wht,16,9,5
5003 datared,red,32,10,28
5004 datacyan,cyn,48,11,159
5005 datapurple,pur,64,12,156
5006 datagreen,grn,80,13,30
5007 datablue,blu,96,14,31
5008 datayellow,yel,112,15,158
5009 dataorange,orn,128,-1,-1
5010 datalt orange,lor,144,-1,-1
5011 datapink,pnk,160,-1,-1
5012 datalt cyan,lcy,176,-1,-1
5013 datalt purple,lpu,192,-1,-1
5014 datalt green,lgr,208,-1,-1
5015 datalt blue,lbl,224,-1,-1
5016 datalt yellow,lye,240,-1,-1
5017 dataillegal,ill,-1,-1,-1
```

F. Complete listing of the sound demonstration program. Note that the listing was printed directly from the program. This was done to avoid typing errors.

```
50 rem vic sound demo by tis
100 ms=40:nn=12:no=3:dimso%(ms,4),oc$(no),no$(nn)
102 nl=5:dimnl$(nl),nl(nl),sc%(nn,no)
105 na=4:fori=1to5:s(i)=36873+i:nexti:vo=12
110 fori=0tonn:readno$(i)
120 forj=1tono:readsc%(i,j)
130 nextj
150 nexti
161 cc$(1)="octave"
162 cc$(2)="note"
163 cc$(3)="length"
171 ac$(1)="read"
172 ac$(2)="write"
173 ac$(3)="compose"
174 ac$(4)="play"
181 oc$(1)="l"
182 oc$(2)="m"
183 oc$(3)="u"
190 nl$(0)="":nl(0)=(1/30)*60
191 nl$(1)="s":nl(1)=(1/16)*120
192 nl$(2)="8":nl(2)=(1/8)*120
193 nl$(3)="q":nl(3)=(1/4)*120
194 nl$(4)="h":nl(4)=(1/2)*120
```

```
195 nl$(5)="w":nl(5)=1*120
200 print"■"
300 print"enter command"
310 inputa$
320 fori=1tona
330 ifac$(i)=a$then400
340 nexti
350 print"legal commands are:"
360 fori=1tona:printac$(i):nexti
370 goto300
400 onigosub1100,1200,1300,1400
410 goto200
1100 input"song name";a$
1110 open1,1,0,a$
1120 ip=0
1130 fori=1to4:input#1,x
1140 ifx<0then1180
1150 so%(ip+1,i)=x:nexti
1160 ip=ip+1:ifip<msthen1130
1170 print"song too long"
1180 close1:return
1200 input"song name";a$
1210 open1,1,1,a$
1215 ifip=0then1230
1220 fori=1toip:forj=1to4:print#1,so%(i,j):nextj,i
1230 print#1,-1:close1
1240 return
1300 ip=0
1350 gosub3000
1360 ifok=1thenreturn
1370 ifip<msthen1350
1380 print"song done. no more room":return
1400 ifip=0thenreturn
1410 fori=1toip:forj=1to3:pokes(j),so%(i,j)
1420 nextj
1425 pokes(5),vo
1430 x=ti+so%(i,4)
1440 ifti<xthen1440
1450 pokes(5),0
1460 x=ti+nl(0)
1470 ifti<xthen1470
1480 nexti
1490 return
3000 cc=1:ok=0:print"■ note";ip+1
3010 gosub4000:ifa$="end"thenok=1:return
3020 ip=ip+1
3040 forcn=1tono
3050 ifa$=oc$(cn)then3100
3060 nextcn
3070 print"error legal ";cc$(cc);" are:"
3080 fori=1tono:printoc$(i):nexti
3090 ip=ip-1:goto3010
3100 cc=2:gosub4000
```

```
3110 fori=0tonn:ifa$=no$(i)then3200
3120 nexti
3130 print"bad note.  legal are:";
3140 fori=1tonn:printno$(i);" ";:nexti:print
3150 goto3100
3200 so%(ip,2)=sc%(i,cn)
3210 cc=3:gosub4000
3220 fori=1tonl:ifa$=nl$(i)then3300
3230 nexti
3240 print"bad time.  legal are:";
3250 fori=1tonl:printnl$(i);" ";:nexti:print
3260 goto3210
3300 so%(ip,4)=nl(i)
3310 so%(ip,1)=0:so%(ip,3)=0
3999 return
4000 print"enter ";cc$(cc)
4010 inputa$:return
5000 datar,0,0,0
5001 datac,135,195,225
5002 datac#,143,199,227
5003 datad,147,201,228
5004 datad#,151,203,229
5005 datae,159,207,231
5006 dataf,163,209,232
5007 dataf#,167,212,233
5008 datag,175,215,235
5009 datag#,179,217,236
5010 dataa,183,219,237
5011 dataa#,187,221,238
5012 datab,191,223,239
```

APPENDIX B

DATA PROCESSING BACKGROUND

A. Information

If you understand certain basic terms and conventions that are used with digital computers, you will find it easier to understand how they operate. Common words, such as "address" and "location", take on specialized meanings when used in reference to the computer. Computer terms and conventions can be divided into two categories: units of information, and storage or retrieval of information.

1. Units . All information processed by a computer, whether data or instructions, can be divided into units of various sizes. In this section, we will describe the forms of the various units of information used by computers as well as the terms used to describe these units.

A computer cannot work directly with decimal numbers, letters, or special symbols such as #, -, +, %, etc. How then does the computer understand these characters since we can see that they are used for both input and output? The internal electronic parts of a computer have only two operating levels or states. A current is flowing or it isn't; a part is magnetized in one direction or another; a switch is either open or closed, etc. Now, we can arbitrarily pick a value for each state that a computer may assume. Since there are only two states, it is customary to pick the values (or numbers) 0 and 1. All computer information, whether instructions, data, numbers, alphabetic letters, or special symbols, may be represented by a combination of 0s and 1s.

Figure B-1 shows how a specific combination of 0s and 1s can represent a specific character. The alphabetic character "M" can be represented by the states of seven electronic circuits.

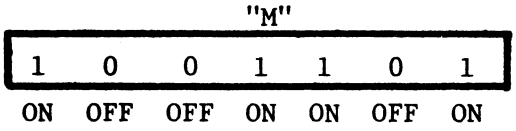


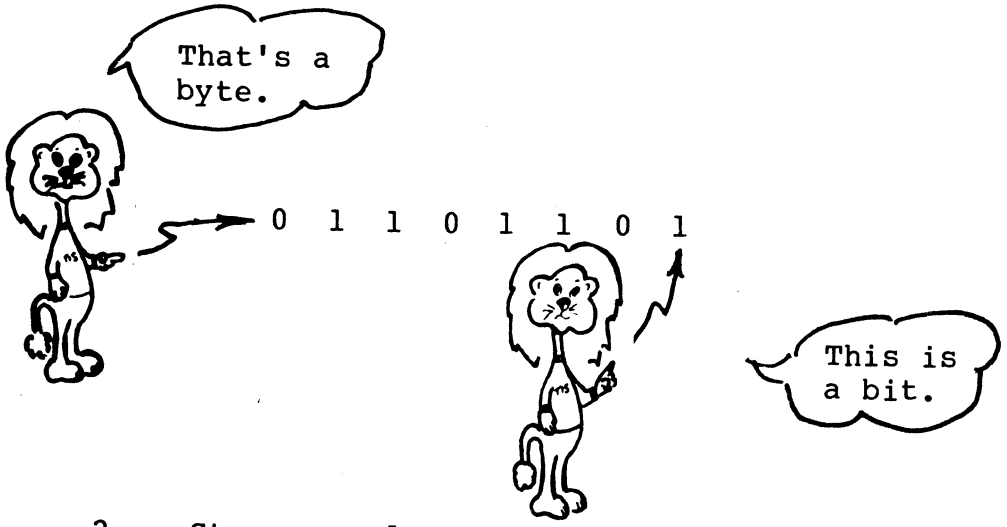
Fig. B-1. Representing a Character.

Each circuit that is on represents a 1, and each circuit that is off represents a 0. The combination shown in the figure (1001101) is a representation of the letter "M". This representation is one of convention. Since there are 128 different combinations for these seven circuits, they can collectively represent 128 unique characters.

Because a computer uses only two digits (0 and 1), they are referred to as binary digits. Thus, a binary digit, or "bit" as it is commonly called, is the smallest unit of information used in a computer. Remember, a bit must always be either a 0 or a 1. Computers cannot store or retrieve individual bits when transferring information to or from memory. Therefore, larger elements of information called "bytes" or "words" are used by the computer.

In many micro computers, eight bits are grouped together and handled as a single element known as a "byte." A single byte might represent a decimal number, a letter, a special symbol, or part of an address. The next largest element of information is a "word." A word is made up of two or more bytes, depending on the design of the computer. Most micro computers use bytes as the largest element of information. A byte is, therefore, the smallest unit of information that is addressable, that is, can be stored or retrieved from memory. A bit, although the smallest unit of information, is not individually addressable. Remember, a byte is 8 bits long.

The example, Fig. B-1, shows 7 bits. The standard ASCII code uses 7 bits to represent characters; the 8th bit is used for error checking. Hence, a character takes a full byte.



2. Storage and Retrieval . Any information to be used by the computer must be stored where the computer can find it when needed. When specific information is needed, it must be located within the storage area and then retrieved.

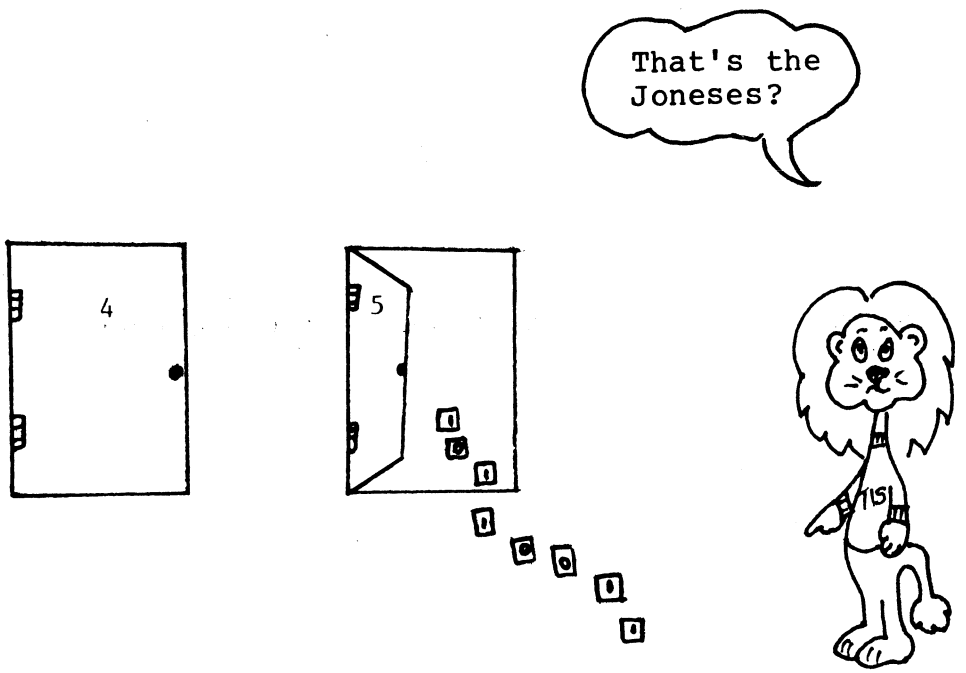
Because main memory can store thousands of bits of information, the memory must be arranged in such a fashion that the central processing unit (CPU) will know where to store information and where to obtain the information it needs to perform a job.

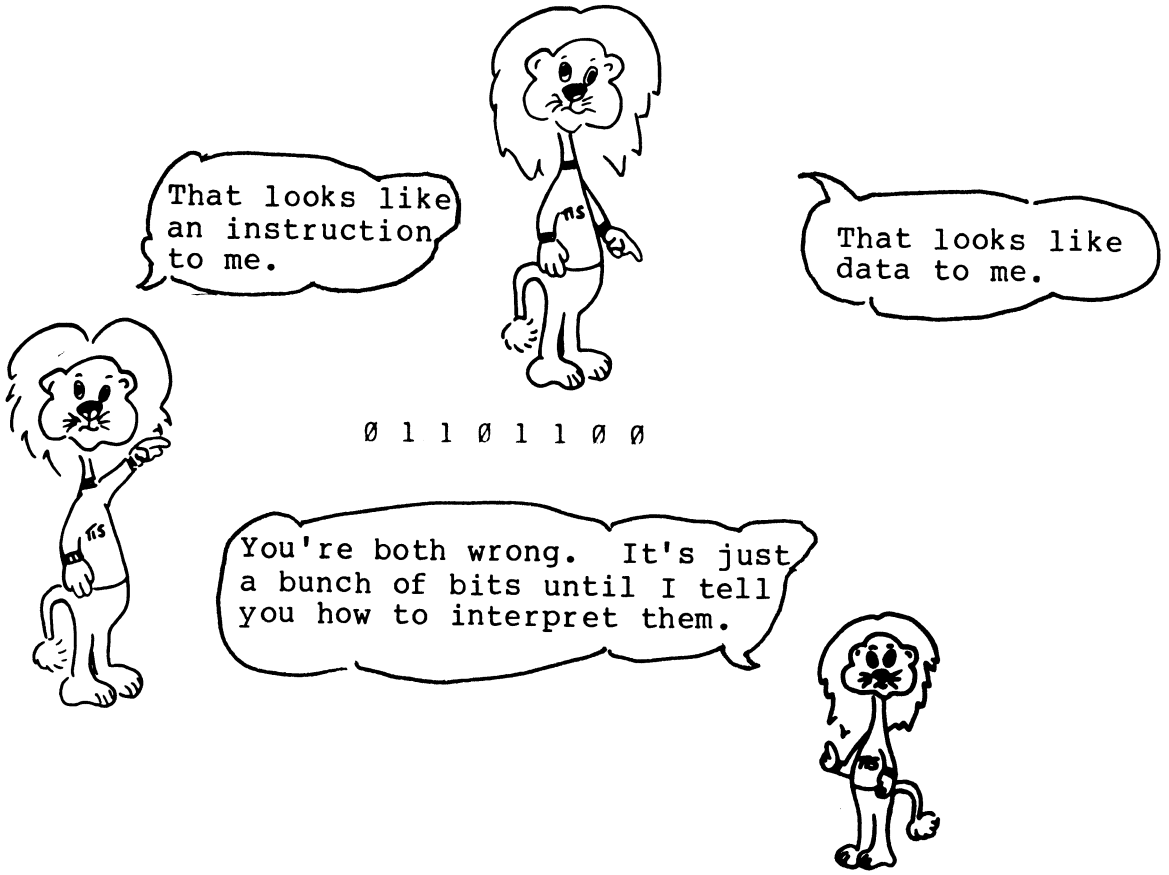
The main memory is made up of thousands of individual cells or memory locations. In our case, these are the individual bytes. Each location is used to store information that the CPU (central processing unit) will use for processing or holding the results of processing. An individual location can only hold one piece of information at a time. Each location in main memory is given a unique address at the time of manufacture. It is important to remember that the address specifies the location of a byte of memory and not the contents of that memory.

When information is stored in a location, it replaces the old contents of that location. However, the contents are not altered when information is retrieved from a location. Once information is stored, it may be read over and over. A location is like a motel room where information lives for a period of time. Since we have many motel rooms, we can assign each room a unique number or address. Now, when we have a request for a room, say for the Jones' information, we can assign the room with address 5. The contents or residents of location 5 is the information belonging to the Jones who have asked for reservations for 5 microseconds.

Memory addresses are numbers that begin at 0 and progress to the highest number required to identify all the locations within a specific memory. Because each location has its own unique address, the CPU can go directly to any location in memory. Although the address of a location is permanent, the contents of a location can be changed at any time.

Before the computer can perform any job, it needs two types of information: the data to be processed and the instructions that tell the computer what to do with the data. Both instructions and data are stored in main memory. The only difference between instructions and data is the way the computer tries to interpret the information in a given byte.

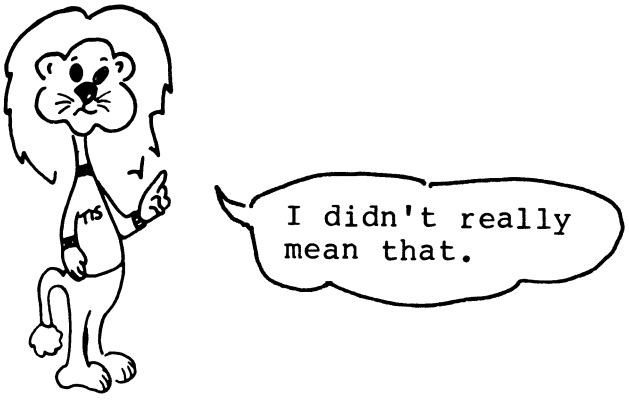



That looks like an instruction to me.
That looks like data to me.
0 1 1 0 1 1 0 0
You're both wrong. It's just a bunch of bits until I tell you how to interpret them.

Computer memories are available in different sizes. Memory size refers to the total number of bytes that are available. The letter "K" is normally used to indicate the size of a particular memory. "K" represents the number 1,024. Thus, a 1 K memory contains 1,024 bytes. It is important to remember that each memory location (byte) is assigned a unique address. A memory of 32K bytes must have 32,768 unique addresses.

B. Symbols

Symbols convey information; the symbol itself is not the information but merely represents it. The printed characters on this page are symbols that convey the writer's meaning when understood.



The meaning of symbols is really one of convention and definition. A symbol may convey one meaning to some persons, a different meaning to others, and no meaning to those persons who do not understand its significance. In order to communicate with the computer, we must use the set of symbols that the computer has been designed to accept and understand. The choice of those symbols and their meanings is a matter of convention on the part of the designers. Many of the symbols used in BASIC are the standard symbols used in everyday writing. A few symbols have been changed to accommodate the terminals that have been used and to avoid confusion with other symbols.

1. Arithmetic Symbols . Table B-1 defines some commonly used arithmetic symbols. The symbols for add, subtract, and divide are self-explanatory and are the same symbols used in normal arithmetic.

Note, however, that the symbol for the multiply operation is an asterisk rather than the conventional arithmetic X. When dealing with computers, the convention is to use the asterisk for the multiplication symbol.

Notice also that the symbol may be used to show a number raised to a certain power. As shown in the table, 2 4 is 16. Raising a number to a power is also referred to as exponentiation because it is identical to placing an exponent after the number. For instance, the expression 2 4 can be read as either "two raised to the fourth power" or as "two with an exponent of four." In either case, the function of the exponent (or power) is the same. It indicates how many times the base number is multiplied by itself. Thus, 2 4 indicates that the number 2 is multiplied by itself 4 times (2 times 2 times 2 times 2).

Table B-1. Arithmetic Symbols

Symbol	Meaning	Example
+	add	Y + 7
-	subtract	Y - 7
*	multiply	Y * 7
/	divide	Y / 7
↑	raise to the power of	Y ↑ 4

2. Relational Symbols . Computers are often used to compare two quantities to determine if they are equal or not equal, or to determine if one quantity is larger or smaller than the other. The symbols used to express these conditions are listed in Table B-2.

Table B-2. Relational Symbols.

Symbol	Meaning	Example	Comments
=	equal to	A = B	A is equal to B
<>	not equal to	A<>B	A is not equal to B
>	greater than	X > 7	X is greater than 7
<	less than	X < 7	X is less than 7
<= =<	less than or equal to	X <= 7	X is less than or equal to 7
>= =>	greater than or equal to	X >= 7	X is greater than or equal to 7

In BASIC, the = sign has two interpretations. In Table B-2 the = sign is used as a conditional symbol: A = B. This means to test for an equality condition, that is, whether the value of A precisely equals the value of B.

The second way in which the = symbol can be used is when writing equations for computer programs. In this context, the meaning of the = sign is interpreted as "is replaced by." For example, the equation : A = B, is read as "the value of A is replaced by the current value of B."

Assume that location A has a 3 stored in it, and assume location B has a 10 stored in it. The equation A = B changes only the contents of A. The location A would now have a 10 and location B would also have a 10.

As you may have noticed, we used the symbols A and B to reference two memory locations and let BASIC assign the specific addresses to A and B. This is known as symbolic addressing.

C. Number Systems

When people process information manually, they deal with alphabetic and numeric characters. However, because computers are binary machines, they represent information internally with 1s and 0s. Therefore, computers must perform mathematical operations using the binary number system and must represent alphanumeric information using binary codes.

To prepare information effectively for computer processing, people must understand how the computer will manipulate the information. This section covers binary numbers. It also presents the octal and hexadecimal number systems, which many people use as a convenient shorter form in place of binary when dealing with computer information.

The four number systems that we will work with are: the decimal system , which we use in our everyday work; the binary system , which digital computers use because it is easy to implement electronically; and the octal and hexadecimal systems , which programmers use to represent binary numbers because they are easier to work with. Hexadecimal (or hex, for short) and octal numbers, however, must still be converted to binary before the computer can process them.

As shown in Table B-3, each number system has a unique base or radix . This base corresponds to the number of digits or unique symbols that are used in that number system. For example, the decimal number system uses 10 digits (0 through 9), and therefore has a radix of 10.

1. Decimal - Binary - Octal -Hexadecimal . In the decimal system, two or more digits are needed to represent any value greater than 9. Whenever a number contains two or more digits, each digit is assigned a specific value called a place or positional value . This place value equals the base of the number system raised to some power . See Appendix B. 1 (Arithmetic Symbols) for an explanation of power. In the decimal system, place values are based on powers of ten (that is, units, tens, hundreds, thousands, etc.).

Decimal to Binary Conversion.

When converting a decimal number to a binary number, the following rules apply:

a. Divide the decimal number by 2 and save the remainder.

b. Divide the quotient from the previous division by 2 and save the remainder.

TABLE B-3. Actual Values

NUMBER SYSTEM	BASE (RADIX)	ACTUAL VALUES (DIGITS)
Decimal	10	0 through 9
Hexadecimal	16	0 through 9 and A - F
Octal	8	0 through 7
Binary	2	0 and 1

c. Continue step b until the quotient is zero.

d. The remainders saved from the division make up the digits of the binary number. The first remainder is the right-most or least significant digit (LSD), and the last remainder is the left-most or most significant digit (MSD).

The example below shows how the decimal number 38 is converted to binary.

NUMBER/BASE	QUOTIENT	REMAINDER
38/2	19	0 (LSD)
19/2	9	1
9/2	4	1
4/2	2	0
2/2	1	0
1/2	0	1 (MSD)

Therefore: 38 = 100110

TABLE B-4

DECIMAL BASE 10	HEXADECIMAL BASE 16	OCTAL BASE 8	BINARY BASE 2
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	8	10	1000
9	9	11	1001
10	A	12	1010
11	B	13	1011
12	C	14	1100
13	D	15	1101
14	E	16	1110
15	F	17	1111

In the binary, or base-two number system, we have the two unique symbols 0 and 1. The place value in this system is a power of two; that is, units, twos, fours, eights, sixteens, etc. Now let's see if Linus can illustrate this:

First, let's number some boxes beginning with 0.

5 4 3 2 1 0

These numbers are going to be used to determine the place value of whatever number system we choose to use.

Why did we start with the number 0?

Because we need a units position and any number raised to the zero power is 1.

$2^0=1$

To determine the value of a number, you must first know in what base it is represented. You then take this base and, using the numbers in the above box as exponents, you determine the place values. Next, you multiply each digit in the number by its corresponding place value. Last, you add all these numbers together. Let's determine the value of the binary number 1011. First, let's recreate the box to determine the exponents.

5 4 3 2 1 0

$2 \uparrow 0 = 1$

$2 \uparrow 1 = 2$

$2 \uparrow 2 = 4$

$2 \uparrow 3 = 8$

$2 \uparrow 4 = 16$

$2 \uparrow 5 = 32$

See how we use the numbers in the box.

The base is two (binary).

Those are the place values.

5 4 3 2 1 0

32 16 8 4 2 1

Multiplying each digit of 1011 by its place value:

1 X 1 = 1

1 x 2 = 2

0 X 4 = 0

1 X 8 = 8

Now adding 8 + 0 + 2 + 1 = 11

The value of the binary number 1011 is equal to decimal 11. The above method works with any number system, if we divide by the appropriate base instead of 2.

The hexadecimal number system sometimes referred to as "hex," has a base of sixteen. A base of sixteen means that the system uses sixteen unique symbols. The symbols that are commonly used are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Table B-4 illustrates the counting numbers for four different number systems.

Notice that it takes more digits to represent a number if the base is small. For example, the number 15 requires 4 binary digits, 2 octal digits, 2 decimal digits, and only 1 hexadecimal digit.

Let's convert the hex number 8AF to decimal. Remember that any base raised to the zero power is always one, and any base raised to the first power is the base itself.

$16 \uparrow 0 = 1$

$16 \uparrow 1 = 16$

$16 \uparrow 2 = 256$

These are the place values.

$F \times 1 = 15 \times 1 = 15$

$A \times 16 = 10 \times 16 = 160$

$8 \times 256 = 2048$

F hex is equal to 15 decimal.

Now add the results:

$15 + 160 + 2048 = 2223$

The hex number 8AF = 2223 decimal.

2. <u>Conversions</u> . It is often necessary to convert from one number system to another.

a. Binary to Octal
b. Binary to Hexadecimal
c. Binary to Decimal
d. Decimal to Binary

The following sections explain the techniques for binary to octal and binary to hexadecimal conversions.

Binary to Octal Conversion

Conversion between octal and binary numbers is fairly straightforward. 8 decimal = 10 octal = 2 ^ 3. This means that every octal digit is equivalent to three binary digits and vice versa (see Table B-5).

The conversion procedure is as follows:

1. Start at the right hand side and arrange the binary digits into groups of three.

2. Substitute, for each group, the equivalent octal digit from table B-5.

Let's convert the binary number 110101111001 to octal.

110	101	111	001	Separate into groups of three.
6	5	7	1	Substitute the octal digits from the table.

$$110101111001_{2} = 6571_{8}$$

The subscripts, 2 and 8, are the base the number is in.

Table B-5. Octal-Binary Equivalents

OCTAL	BINARY
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Binary to Hexadecimal Conversion

Converting between hex and binary is also straightforward since 16 is another power of 2; namely, 16 = 2^4. This means that you can use the same procedure you used above except that you arrange the binary number into groups of four. Let's use Table B-6 and convert the same binary number (110101111001) into hexadecimal.

```
1101     0111     1001          Separate into groups
                                of four.
  D        7        9
                                Substitute the hex
                                digit from the table.

110101111001  = D79
              2      16
```

Table B-6. Hexadecimal - Binary Equivalents

HEXADECIMAL	BINARY
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

D. Character Codes

The method or system used to represent data is known as a code. In the computer, the code relates data to a fixed number of bits. If you use only a single bit to represent each symbol, you are limited to two choices: one symbol for the "ON" state and one symbol for the "OFF" state. This would severely limit our ability to communicate. If you add another bit, you double the number of characters that can be represented. You have two choices with one bit: 0 or 1. You have four choices with two bits: 00, 01, 10, or 11. If you add another bit, you again double the number of choices: 000, 001, 010, 011, 100, 101, 110, and 111. It can be shown that for n bits, the number of choices (or characters) is 2 n.

The American Standard Code for Information Interchange (ASCII) is a 7-bit code that has been established as a standard by the computer industry. The VIC uses a modified version of ASCII.

E. Logical Operations

Logical operations can be used to select options, check parameters for consistency, and control the execution flow of your program.

Exercise: Determine what numeric values the VIC uses to represent true and false.

```
T:  NEW
T:  100 PRINT "ENTER VALUE";
T:  110 INPUT A
T:  200 IF A THEN PRINT "A IS TRUE WHEN A=";A: GOTO 100
T:  210 PRINT "A IS FALSE WHEN A=";A: GOTO 100
T:  RUN
R:  ENTER VALUE?
T:  1
R:  A IS TRUE WHEN A=1
R:  ENTER VALUE?
T:  0
R:  A IS FALSE WHEN A = 0
R:  ENTER VALUE?
T:  .001
R:  A IS TRUE WHEN A = 1E - 03
R:  ENTER VALUE?
T:  -1
R:  A IS TRUE WHEN A =-1
R:  ENTER VALUE?
```

These results show that the VIC interprets zero as false and non-zero as true. However, just any non-zero value will not work properly for true. When NOT is covered, you will find that true must be represented with the non-zero value -1. Type 1E40 to end this program.

Exercise: Determine what string values the VIC uses to represent true and false.

```
T:  NEW
T:  100 PRINT"ENTER STRING";
T:  110 INPUT A$
T:  200 IF A$ THEN PRINT "A$ IS TRUE WHEN A$=";A$:GO TO 100
T:  210 PRINT "A$ IS FALSE WHEN A$=";A$:GO TO 100
T:  RUN
R:  ENTER STRING?
T:  ABC
R:  A$ IS TRUE WHEN A$=ABC
R:  ENTER STRING?
T:  ""
R:  A$ IS FALSE WHEN A$=
R:  ENTER STRING?
T:  "b"
R:  A$ IS TRUE WHEN A$=
R:  ENTER STRING?
```

This shows that the null string ("") is interpreted as false. Any string with at least one character in it is considered true. Type 'RETURN' and 'STOP' simultaneously to end this program.

1. AND Function . The AND function gives a true output only when both inputs are true.

The statement:

Tim AND Bill own computers.

is true only if both Tim and Bill do own computers.

A chart can be built to describe the operation of any logic function. This chart lists every possible combination of input values (binary 1s and 0s) and the output resulting from each combination of inputs. Some people substitute the words "true" and "false" for the values "1" and "0" when using a chart of this type. That's why this chart, or table, is commonly referred to as a "truth table" even though 1s and 0s are often used.

Table B-7. Two-Input AND Function - Truth Table

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Table B-7 is a truth table for a two-input AND function. It lists <u>every</u> possible combination of input values and shows the appropriate output for each combination of inputs. Notice that there is an output (1) only when <u>both</u> inputs are <u>true</u> (or 1s). Thus we can say that "C" (the output) is true when inputs "A" and "B" are true.

Another method of describing the AND function is to use a mathematical formula. Using a formula provides a precise shorthand notation for easy representation of digital logic. Let's go back to the two-input AND function. You know that the output ("C") is true only when both inputs ("A" and "B") are true. Therefore, you can express this relationship by using the formula:

C = A AND B

Exercise: Display AND Truth Table.

```
T:  NEW
T:  90 PRINT "INPUT OUTPUT"
T:  95 PRINT " A  B  C "
T:  100 FOR I= 0 TO 1
T:  110 FOR J = 0 TO 1
T:  120 PRINT I;J;I AND J
T:  130 NEXT J
T:  140 NEXT I
T:  RUN
R:  Table B-7 should be displayed.
```

2. <u>Inclusive OR Function</u> . The inclusive OR function provides a true output if either one input or the other input or both inputs are true. This function produces a true output (binary 1) when at least one input logic condition is satisfied, as indicated by a binary 1 on one of the input values. Thus, we can say that 1 OR 1 = 1.

The statement:

Don OR Dave can program.

is true if either Don or Dave (or both) do know how to program.

This function is called an inclusive OR because any one or more true inputs will provide a true output. In other words, if there were four input values, a true input on any one of the values 1 through 4, inclusive, would provide a true output. Whenever the term "OR" is used by itself, it refers to the "inclusive OR".

Table B-8 is a truth table for a two-input inclusive OR function. It lists all possible combinations of input values and shows the output for each combination of inputs.

A mathematical formula can also be used to represent the inclusive OR function. For example, operation of a two-input inclusive OR can be expressed by the formula:

C = A OR B

Exercise: Display OR Truth Table. Use the program for AND with one modification.

```
T:  120 PRINT I;J; I OR J
T:  RUN
R:  Table B-8 should be displayed.
```

TABLE B-8 Two-Input OR Function - Truth Table

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

3. NOT Function (Inverter) . The purpose of the NOT function is to convert an input value to its opposite (complementary) state. The NOT function is often called an inverter because it inverts the input value. For example, if the input signal is one, the NOT inverts it to a zero, and vice versa.

The statement:

Dick is NOT an engineer.

is false if he is an engineer and true otherwise.

Table B-9 is a truth table for the NOT function. Notice that there are only two possible inputs to the function.

We can also use a mathematical formula to represent the NOT function. We express the NOT function as

C = NOT A

Exercise: Display NOT Truth Table. It may seem reasonable to simply modify the program for the AND and the OR truth tables. This will not work because the NOT truth table (Table B-9) shows that NOT true must be false and NOT false must be true.

You have been using 1 for true and 0 for false. Since NOT false should be true, try:

```
T:  PRINT NOT 0
R:  -1
```

Not 0 is -1. Therefore, you cannot use 1 for logical value true. To double check, NOT true must be false.

```
T:  PRINT NOT -1
R:  0
```

The VIC uses zero to represent false and -1 to represent true.

Table B-9. NOT Function - Truth Table

INPUT	OUTPUT
A	C
0	1
1	0

```
T:  NEW
T:  90 PRINT "INPUTbbbOUTPUT"
T:  100 FOR I=0 TO -1 STEP -1
T:  110 PRINT ABS(I), ABS(NOT I)
T:  120 NEXT I
T:  RUN
R:  Table B-9 should be displayed.
```

F. Relational Operations

The relational operations determine if the relation holds and assigns the value true (-1) or false 0.

Exercise: Determine what the relational operators generate as a result.

```
T:  A=1:  B=2:  X=A<B:  PRINT X
R:  -1
T:  PRINT A>B
R:  0
T:  PRINT A<B
R:  -1
T:  PRINT A=B
R:  0
```

Using -1 for true and 0 for false shows that the result of a relational operation is the logical value of the relation.

In the section on memory organization, you learned that the computer cannot manipulate individual bits separately. The smallest addressable unit is a byte. The logical operations work on each bit in the 8-bit byte in parallel. The VIC treats two 8-bit bytes as the unit for logical operations.

Exercise: Determine the maximum size of quantities used in logical operations.

```
T:  NEW
T:  A=1:B=2
T:  X=32767: PRINT(A<B) AND X
R:  32767
T:  X = 32768 PRINT (A<B) AND X
R:  ?ILLEGAL QUANTITY
R:   ERROR
```

Logical operations are only permitted on 16-bit (or less) quantities.

These products are also available from TIS. Write for your free copy of our current catalog

TOTAL INFORMATION SERVICES INC.
P.O. BOX 921
Los Alamos, NM 87544

VIC SOFTWARE

SAVE typing time!
The complete color and sound demonstration programs (p115-118) on cassette only $7.95+ $1.00 first class postage.

BOOKS PET/CBM

Learn more about COMMODORE PET/CBM! These well-written TIS books provide a comprehensive introduction to many aspects of computing with exercises and sample programs throughout. These documents are specifically designed to complement the PET/CBM computers—they even explain the most up-to-date differences in the ROM features of the PET 2001 and the new CBM including BASIC 4.0.

Understanding Your PET/CBM,
Volume I: Basic Programming
248-page softcover book that combines five of the successful TIS workbooks. In addition, discussions for both ROMs are integrated into the text, there is a ROM Summary Chapter, and a comprehensive index to all topics. Linus, the friendly TIS lion, is featured throughout.

PET Graphics
52 page softcover booklet that will help you use one of the most powerful features of the PET/CBM

Contents

Explains the use of cursor control and special graphics symbols to draw plots, histograms, and sketches. Describes how to use POKE to directly display information on the TV screen.

PET/CBM Software

These TIS software packages offer practical, easy-to-use programs that will greatly increase the usefulness of your COMMODORE PET/CBM. Each package includes a cassette or diskette of the source program, a source listing, and a complete manual of instructions and step-by-step examples showing how to use the program.

SW-1 Mailing List Program

This program allows you to add, delete, correct, display mailing lists entries. It requires a minimum 8K PET/CBM with one cassette. Entries are stored on cassette or diskette and can be printed on your hard copy device.

SW-2 Checkbook Program

This program has many checkbook and budget applications: assists in balancing a checkbook; selects and lists checks by person, purpose, or date; sums checks by category or person.

SW-3 Accounts Program

This program allows you to create a data base for company names, addresses, invoice and purchase order numbers, and amounts of purchase. Locates information on specific companies, determines amounts owed, lists past-due accounts.

SW-4 Micro Text Editor

A cassette-file, line-oriented text editor that allows insertion, deletion and modification of upper- and lower-case alphabetic and numeric strings. Modification is done with cursor control keys. Useful for maintaining data files.

SW-5 Calendar Program

This program enables you to keep track of appointments in a busy office, schedule your social engagements, etc. It is an all-purpose datebook complete with events, dates, times, and notes.

ABOUT THE AUTHOR

Mr. Schultz has been a computer professional for twenty years. His background includes teaching courses in computing science, writing applications and systems programs, supervising software development projects, and doing research on minicomputer-based intelligent terminals. For the past fourteen years he has been employed by the Los Alamos National Laboratory where he has worked with computers of various sizes. He has been responsible for programmer and operator training. Currently he is working on realtime software for a mini- and microcomputer network system for controlling a large accelerator. Having started early in the personal computer field with the construction of a MARK-8 in 1973, he later obtained a KIM-1, then an early serial numbered PET, a CBM and now a VIC. He is the co-author of Understanding Your PET/CBM Volume 1 Basic Programming and Understanding Your C1P/C4P a Workbook of BASIC Exercises and theauthor of PET Graphics. He brings a professional approach to using a personal computer.

The familiarity with such a broad range of computers and computer users has produced a rare balance of parcticality and professionalism in this "how-to" manual which should prove to be refreshingly useful to the VIC personal computer user.